## Java and DevOps Training

# Project 2 Report

### In-Memory Database

**Prepared by:**

**Mamoun Hayel Abu Koush**

mamounhayel@gmail.com /in/MamounHayel

# Table of Contents

# 1 Introduction

## 1.1 Introduction

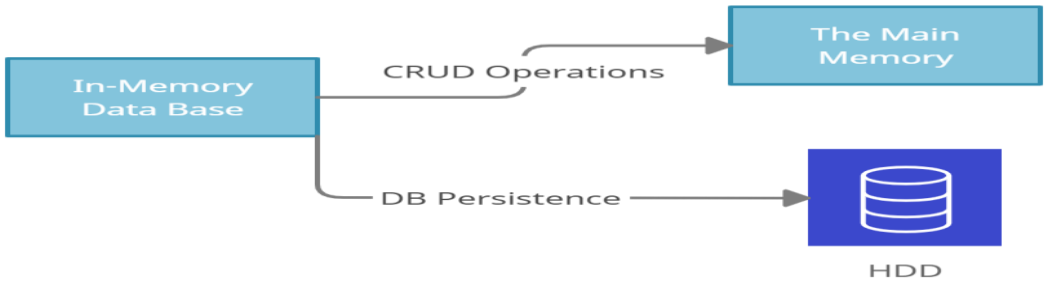


*Figure 1. In-Memory DB with Persistence.*

In this report I will describe how I built and designed the in-memory database Web Application; it was a pretty challenging project. The DB is multi-threaded and supports all CRUD operations. no unauthorized users can access the database as role-based access control (RBAC) approach is applied to restrict system access to authorized users only. And as a database I created a Library database.

Multiple users with different roles and responsibilities can use the system at the same time and preform CRUD operations. the dataset will be stored on the main memory for faster access, and to keep the database persistent on HDD it will persist each operation on disk in a transaction log (CSV file) so if the machine is down the data won't be lost, and to make the design easier to understand, maintain, and extend, SOLID principles, Design Patterns and Effective Java were applied, to be honest it was more about researching and learning more than implementing.

The system will interact with 2 databases, the in-memory DB and the Users database, to clarify the context and the boundaries of the system and what is the relationship of the system and the entities I have designed the following context diagram

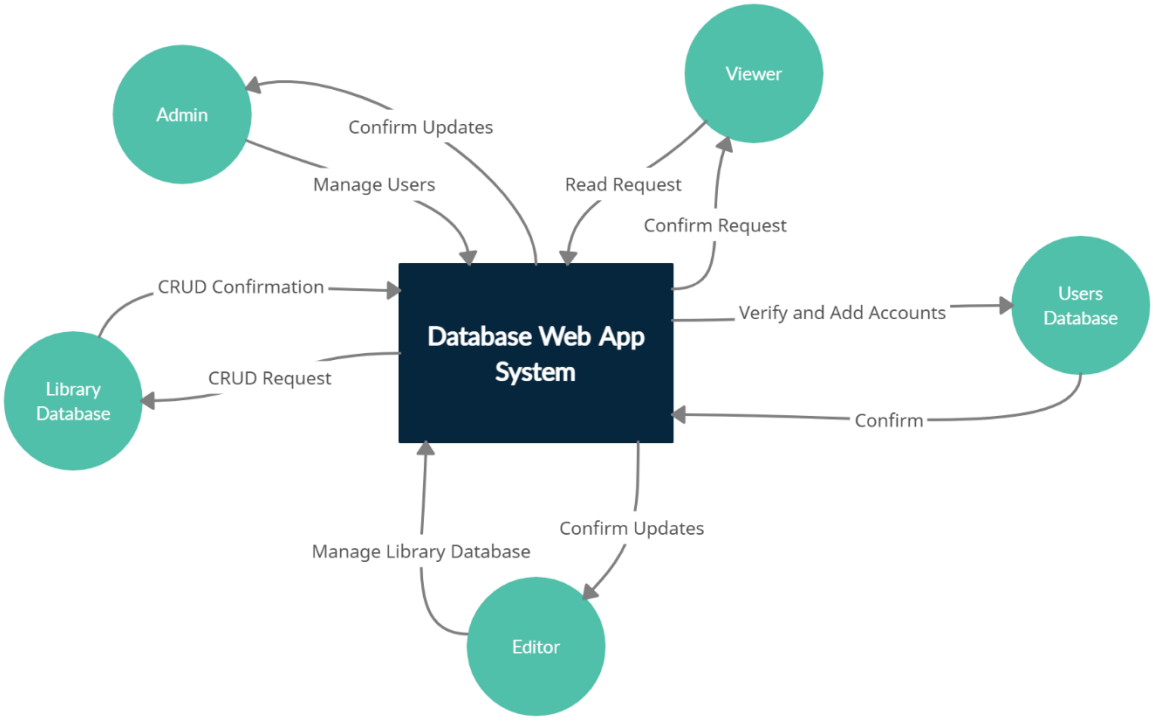


*Figure 2. System Context Diagram.*

The read queries will not slow down the Web App, because they still hit the main memory or cache. But in the update case, each update will not only be applied to in-memory or cache but also persisted on disk in the fastest way possible, the Library In-memory database wont use disk for non-change operations.

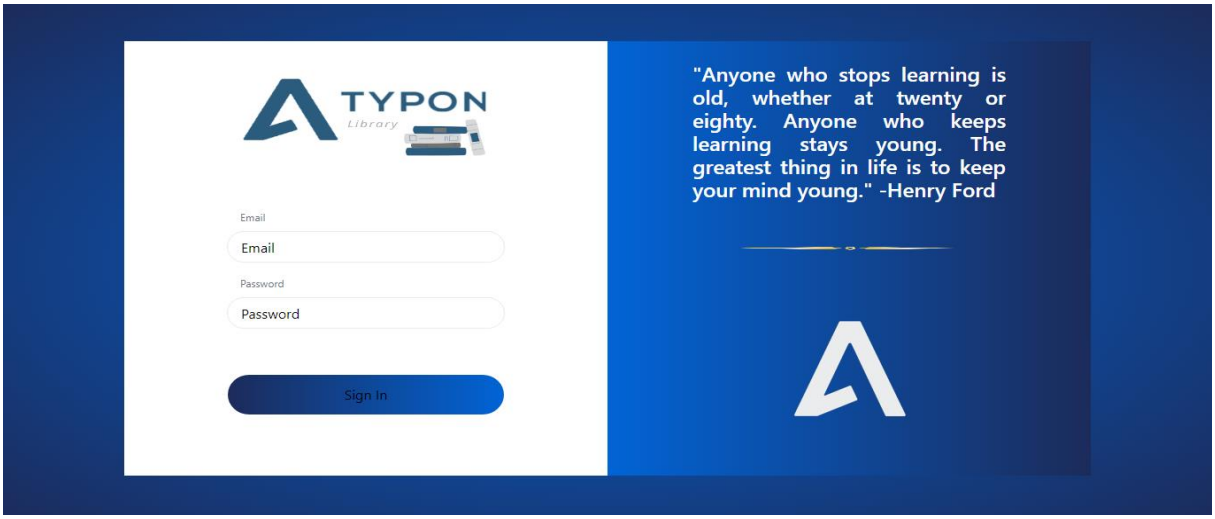


*Figure 3. Atypon library web project.*

# 2 Design and Analysis

## 2.1 Introduction

In this chapter I'll talk about the design and Analysis of the DB System, I obtained the system requirements from the assignment file, Brainstorming, and studying analogous systems, The following subsections introduces more details about the analysis and requirement collections.

## 2.2 Stakeholders

The system would be used by multiple users, members are assigned a user role that determine the privileges that can be granted to the user:

- Admin: can add and delete Users.
- Editor: can view and edit data in the DB.
- Viewer: can only view records in the DB, Viewers can't create, edit, delete any record on the DB.

## 2.3 Functional Requirements

- Authentication of a user when he/she tries to log into the system.
- The system shall allow Admin users to add and delete users.
- The system shall allow editor users to perform all CRUD operations.
- The system shall allow Viewer users to perform read operations only.
- The system shall inform the user with confirmation update about his last CRUD operation. (i.e., book updated successfully).
- The system shall store and persist the data into a database file.

## 2.4 Non-Functional Requirements

- The system shall provide the user with a friendly menu.
- The system shall have high security. For example, user passwords are not stored as plain text in users DB, they shall be hashed first and then salt is added, only the hashed password and salt are stored.
- The system shall store and persist the data into a database file.
- The database server shall be multithreaded.
- To give the maximum performance caching should be applied.
- Non-blocking data structures should be used.

## 2.5 Use Case Diagrams

- Admin use case:

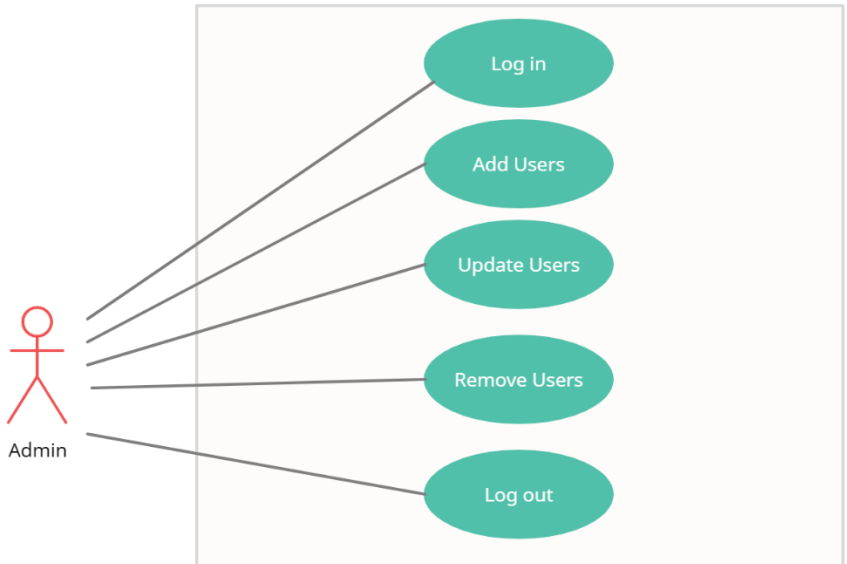


*Figure 4. Admin Use Case Diagram.*

- Viewer use case:

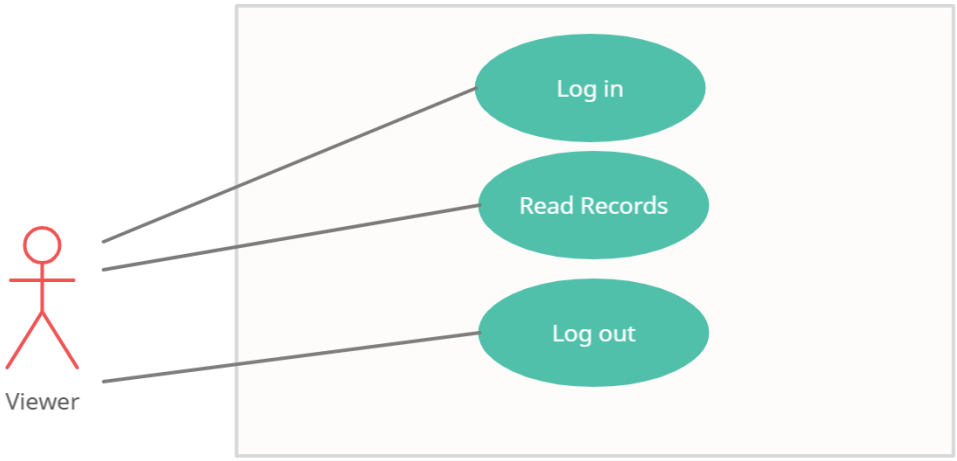


*Figure 5. Viewer Use Case Diagram.*

- Editor Use case:

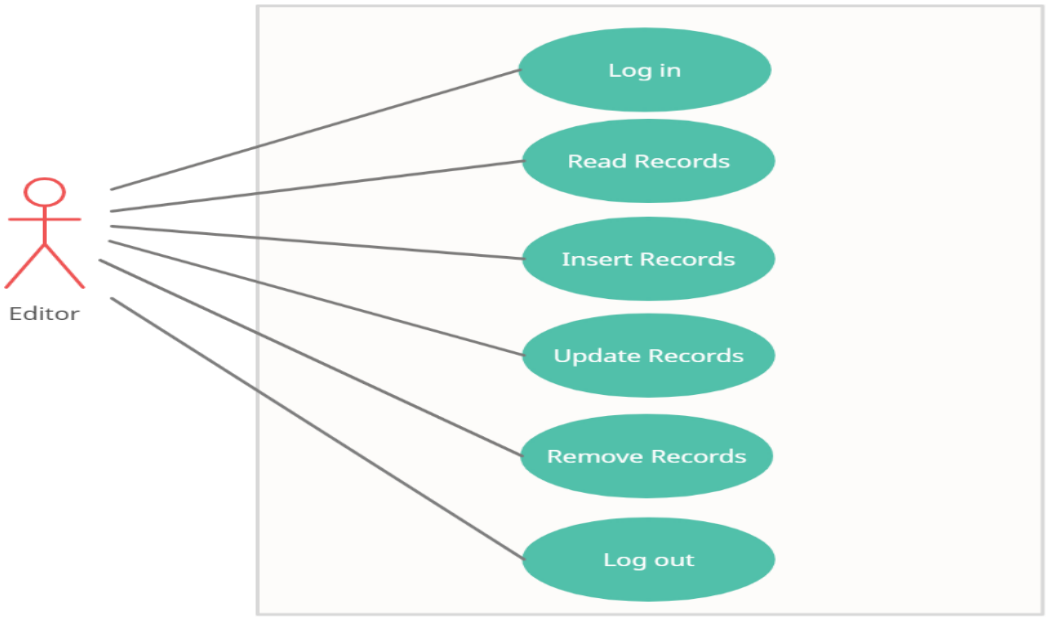


*Figure 6. Editor Use Case Diagram*

## 2.6 Database Tables

- Users Table:

| Field Name | Data Type | Allow Null | Description |
|---|---|---|---|
| **Fname** | String | No | User first name. |
| **Lname** | String | No | User last name. |
| **Username** | String | No | Unique username (Primary Key). |
| **HashedPassword** | String | No | Hashed value = SHA512 (Password + Salt value). In spring version of project, it will be store as a **SCrypt**. |
| **Salt** | String | No | Salt value. No need for it in Spring Version. Needed only in traditional Servlets/Jsp. |
| **Role** | Enum | No | User Role (Admin,Editor,Viewer) |

*Table 1. Users Database Table.*

- Books Table:

| Field Name | Data Type | AllowNull | Description |
|---|---|---|---|
| **ID** | Int | No | Unique Book Number - ISBN (Primary Key). |
| **Name** | String | No | Book Name. |
| **Author** | String | No | Book Author name. |
| **Subject** | String | No | Book Subject (I.e., History, Biography) |
| **Publisher** | String | No | Book Publisher Name. |
| **Publication Year** | String | No | Book Publication Year |

*Table 2. Books Database Table.*

- Quotes Table:

| Field Name | Data Type | AllowNull | Description |
|---|---|---|---|
| **ID** | Int | No | Unique Quote Id. (Primary Key). |
| **BookId** | Int | No | Book Id (Foreign Key), on delete cascade. |
| **Quote** | String | No | Selected quote from the referenced book. |

*Table 3. Quotes Database Table.*

# 3 Implementation

## 3.1 Introduction

In this chapter ill talk about how I implemented the system, the protocol between the client and server, DevOps, how design patterns are used. Data structures implemented, how did I tried as much as possible to abide by the rules of clean code, SOLID principles, effective java and ACID criteria.

## 3.2 DevOps

DevOps is all about implementing the best practices with the help of tools To Provide Continues delivery with high quality software, learning about DevOps was the most satisfactory learning process I have ever gone through. Jez Humble (co-author of the DevOps Handbook) describes it as "DevOps is not a Goal, But a never-ending process of continual improvement". The more I learned about DevOps and the tools used, the more I felt hungry for more and more learning.

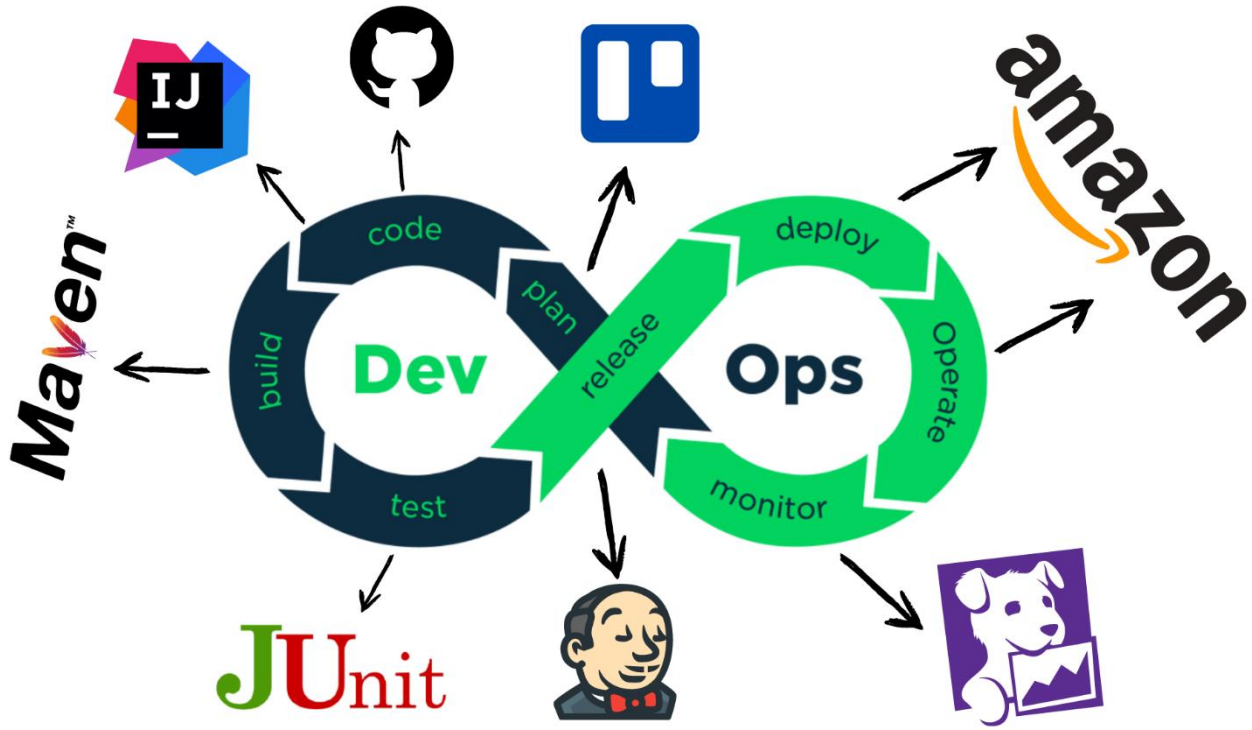


*Figure 7. DevOps Pipeline.*

## 1. Trello

In the first stage of DevOps cycle, I used my Trello account to create a Kanban board to plan my project before starting to work on it, using it helped me a lot on managing my tasks and my schedule. if we had to do this project in teams using Trello would be a very good experience and a great way to share knowledge between the DevOps Team. I also used my notepad for sketching and brainstorming.

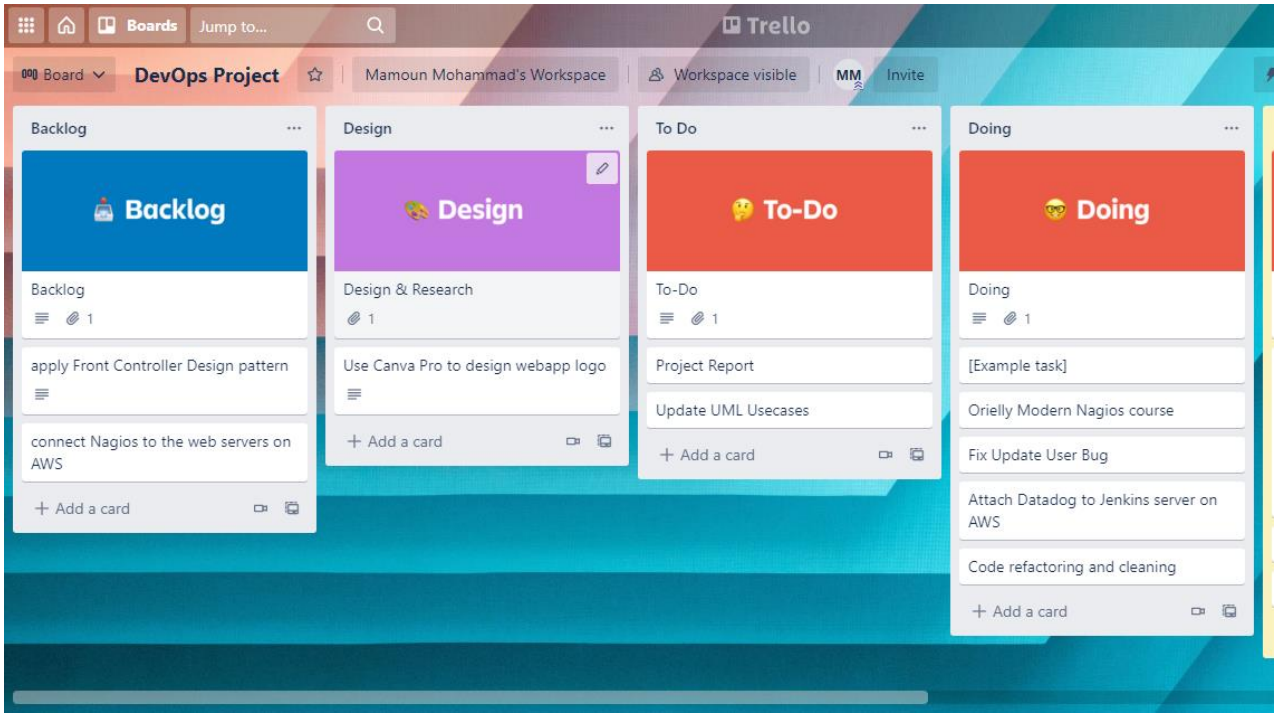


*Figure 8. Kanban Board.*

## 2. IntelliJ Ultimate

IntelliJ Ultimate is one of the best IDEs for java development, its an excellent software. I have used eclipse and its not even close to IntelliJ in Terms of performance.

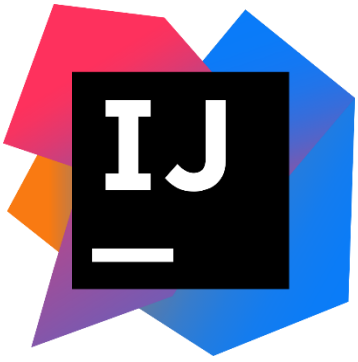


*Figure 9. IntelliJ Logo*

## 3. GitHub

For me whenever someone mentions Version control the first thing that comes to my mind is GitHub, its known to everyone that GitHub is the best version control system, it's the best platform to store our code without any disturbance.

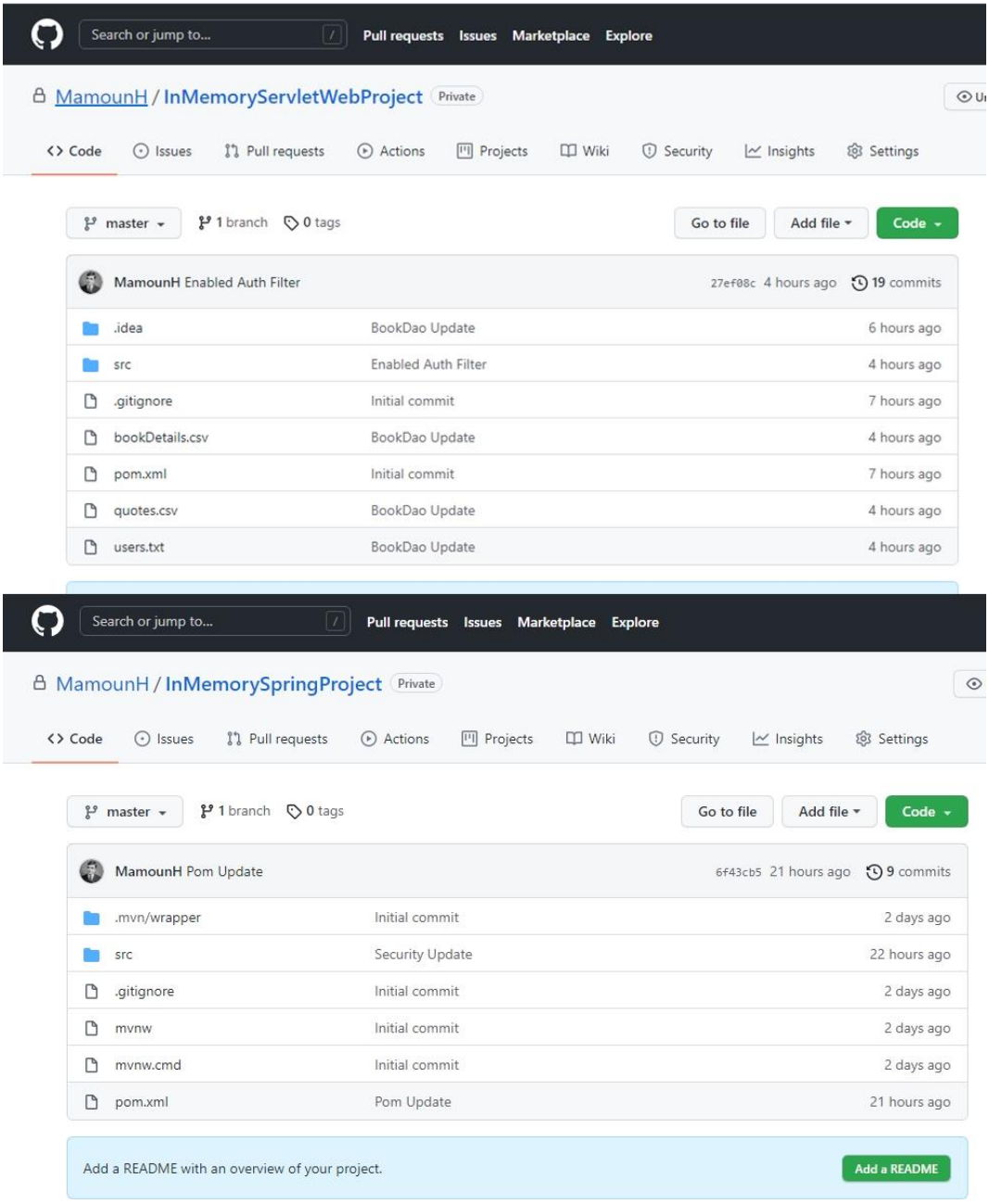


*Figure 10. In-Memory Servlet/Jsp and Spring Web App on GitHub.*

## 4. Maven

Maven is a powerful build automation tool for Java projects, it can be used for any java project. It provides an easy way to build the project in which the complex details are hidden with the help of dynamic downloading of plug-ins and libraries from the maven repository website.


*Figure 11. Maven Logo.*

## 5. Junit

In the testing phase of DevOps cycle, I used Junit framework, once a build succeeds and is deployed to the testing phase a series of unit testing are preformed, also Manual testing is performed to test all aspects of the web application.


*Figure 12. JUnit Logo.*

## 6. Jenkins

As Jenkins provides CI/CD, I think this one of the most important phases in DevOps pipeline, when we reach this stage that means that a build is ready and safe for deployment for the next stage. Since each code has passed a bunch of manual and unit testing. So, we will be confident about the deployment because the likelihood of issues or bugs are very low.

I used AWS EC2 instance as my Jenkins server and linked it to my GitHub repositories. So, whenever a change is made and committed to GitHub from my local machine the whole pipeline will be triggered.

I have configured Jenkins with many plugins, i.e., GitHub, Maven, Datadog.

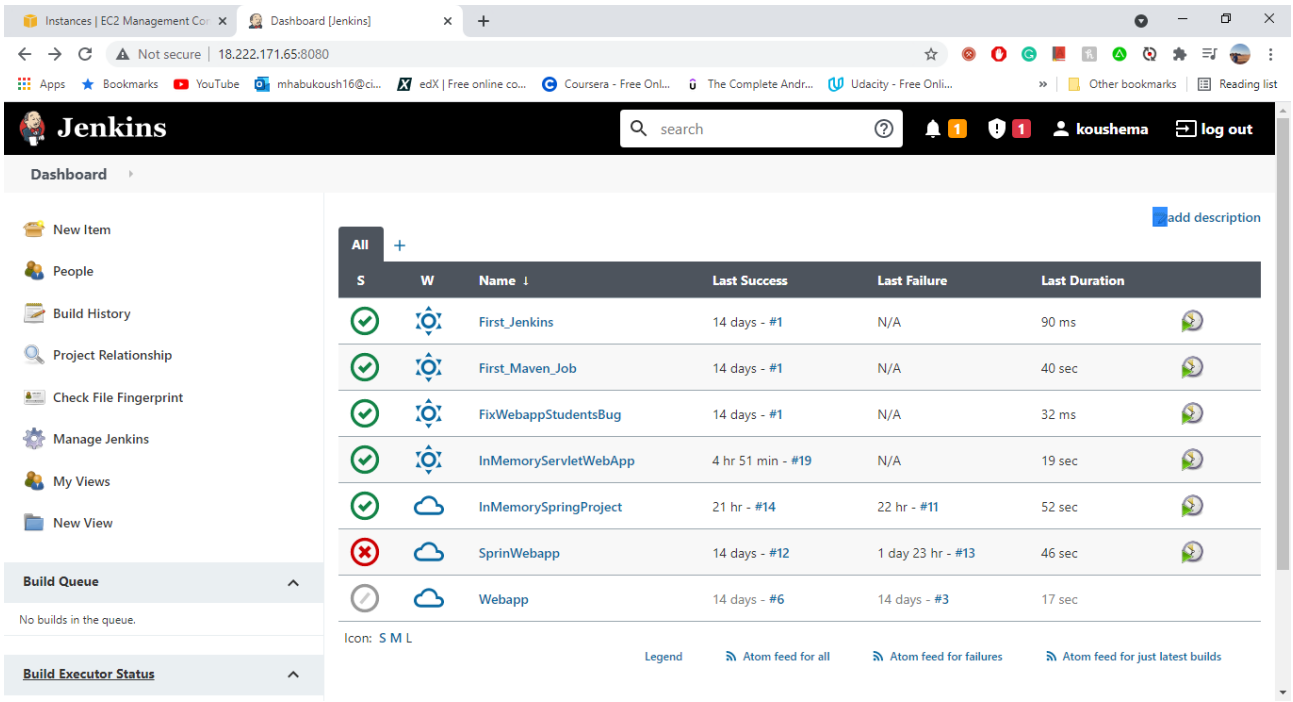


*Figure 13. Jenkins Dashboard.*

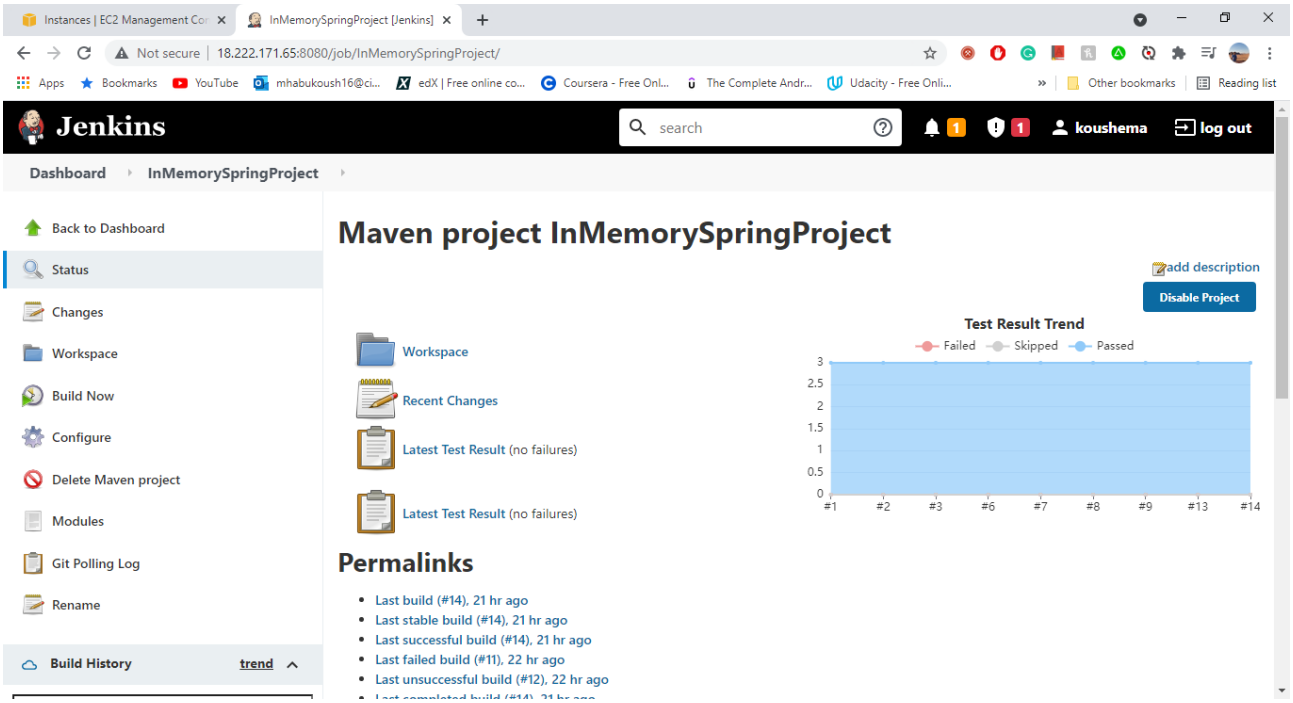


*Figure 14. In Memory Spring Project.*


*Figure 15. Jenkins Server Status on AWS EC2 Instance.*

## 7. Amazon Web Services

When a build is ready to be deployed, Jenkins server will deploy it to AWS, I used here two EC2 instances one for Traditional Web application using Servlets/Jsp and the other is for the spring version of the project. Since we already know that the tests were built successfully, we do not have to worry a lot if the production release will be facing some issues.

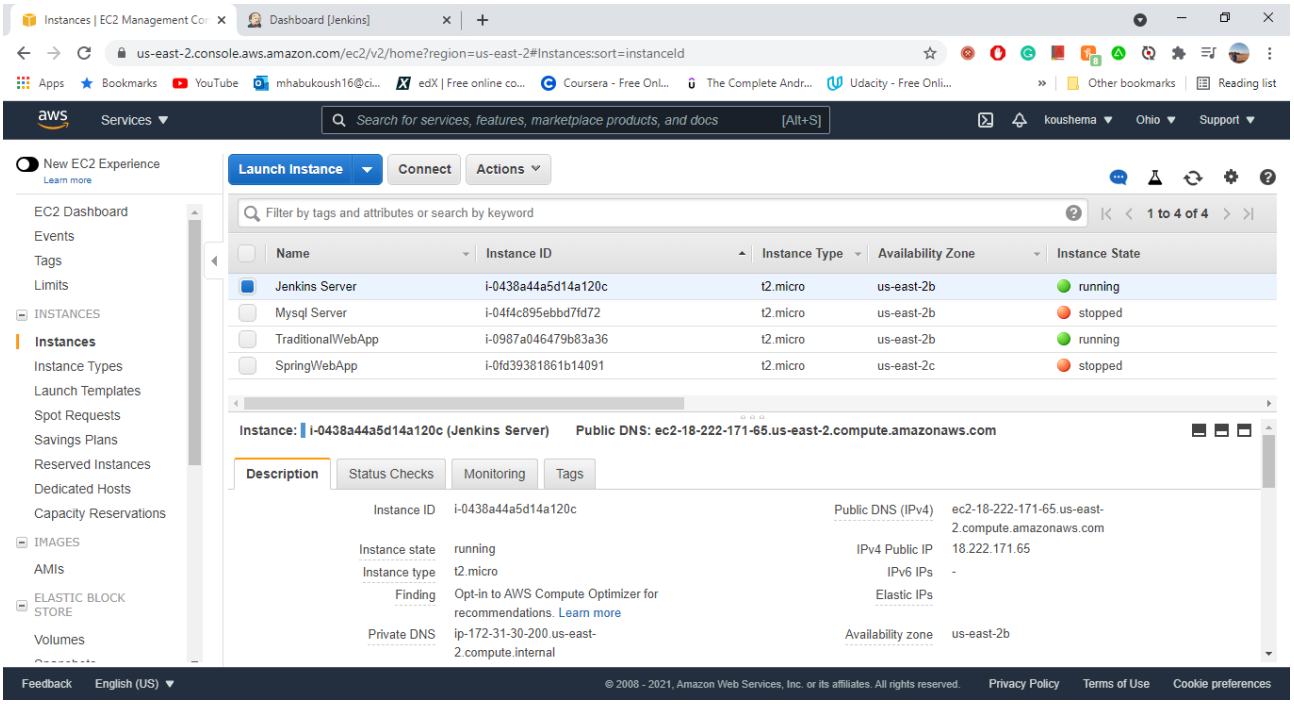


*Figure 16. AWS EC2 Instances.*

# 8. Datadog

In the final phase of the DevOps cycle, I used Datadog, first I defined an access policy on my AWS account to give Datadog authorization to read data from my account and then I assigned an IAM role for Datadog to my Datadog account thus giving read access to monitor the instances. Datadog will be collecting data and providing analytics that will be very helpful for future decisions i.e., Performance, Errors and more.

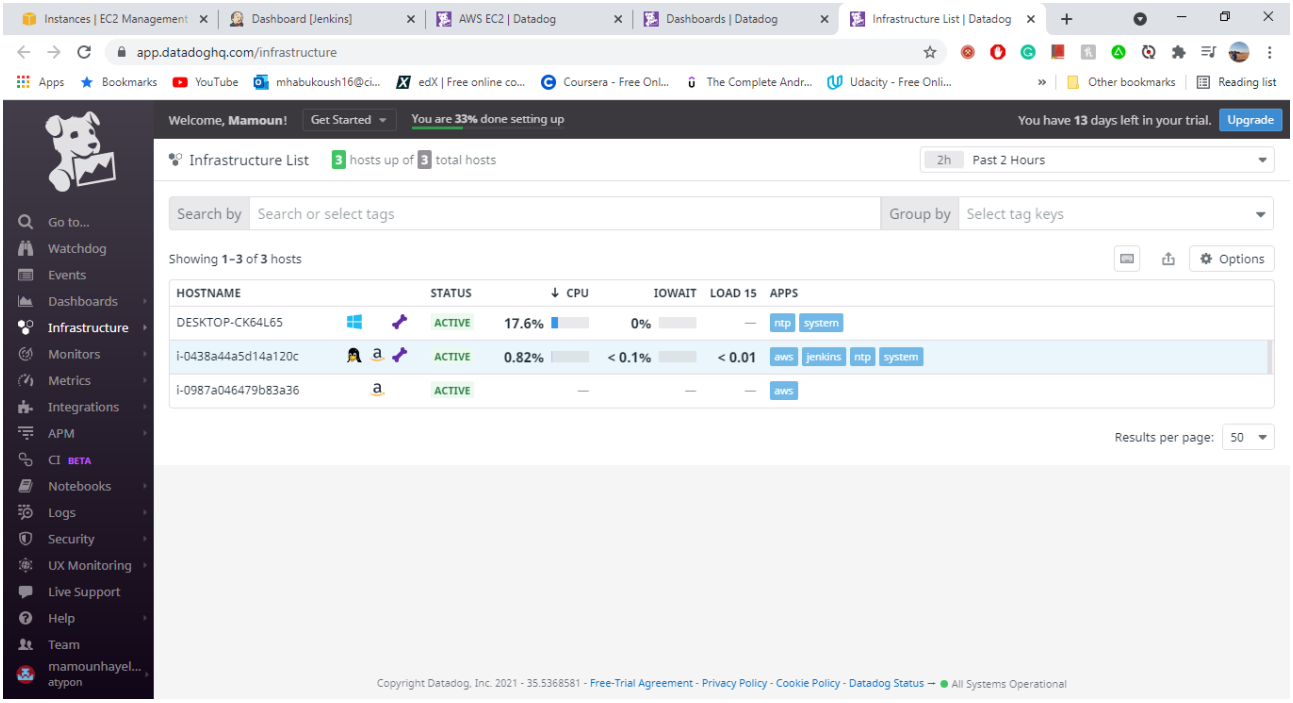


*Figure 17. Datadog Infrastructure List.*

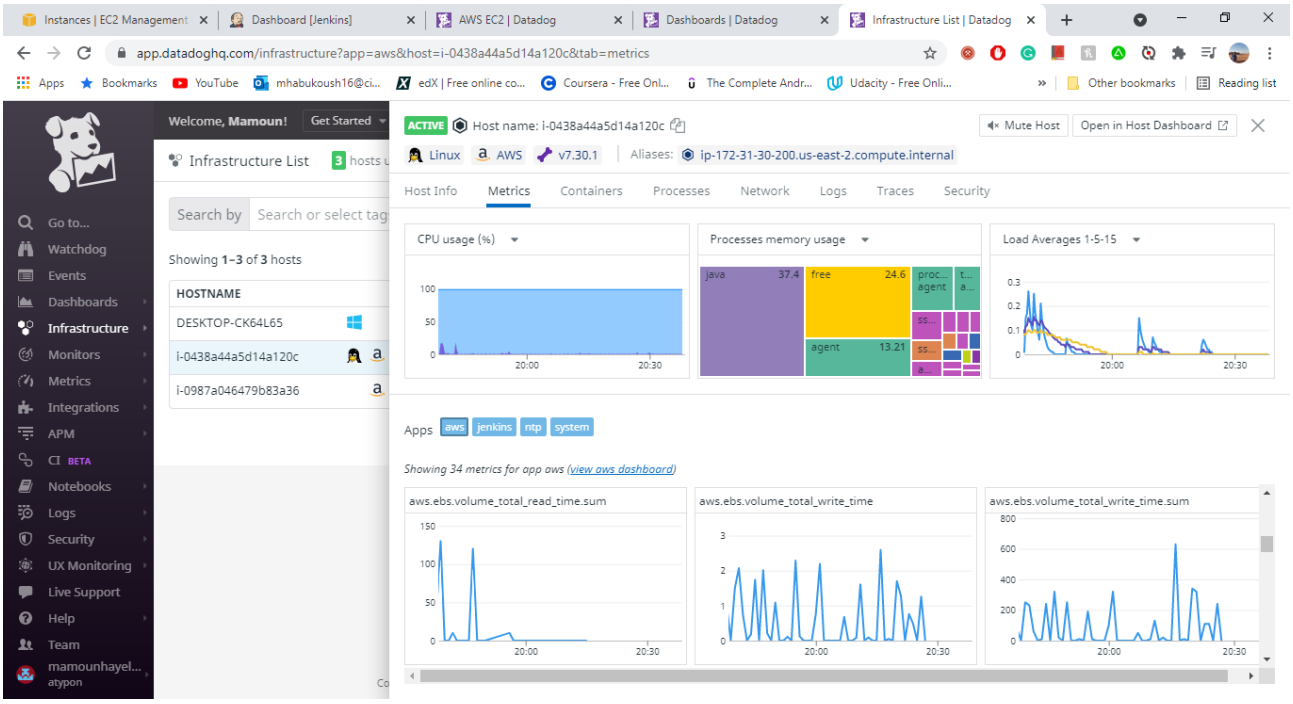


*Figure 18. AWS EC2 instance analytics.*

and also, to do more observation and to monitor the pipeline itself I configured and connected the Jenkins server itself to Datadog itself.

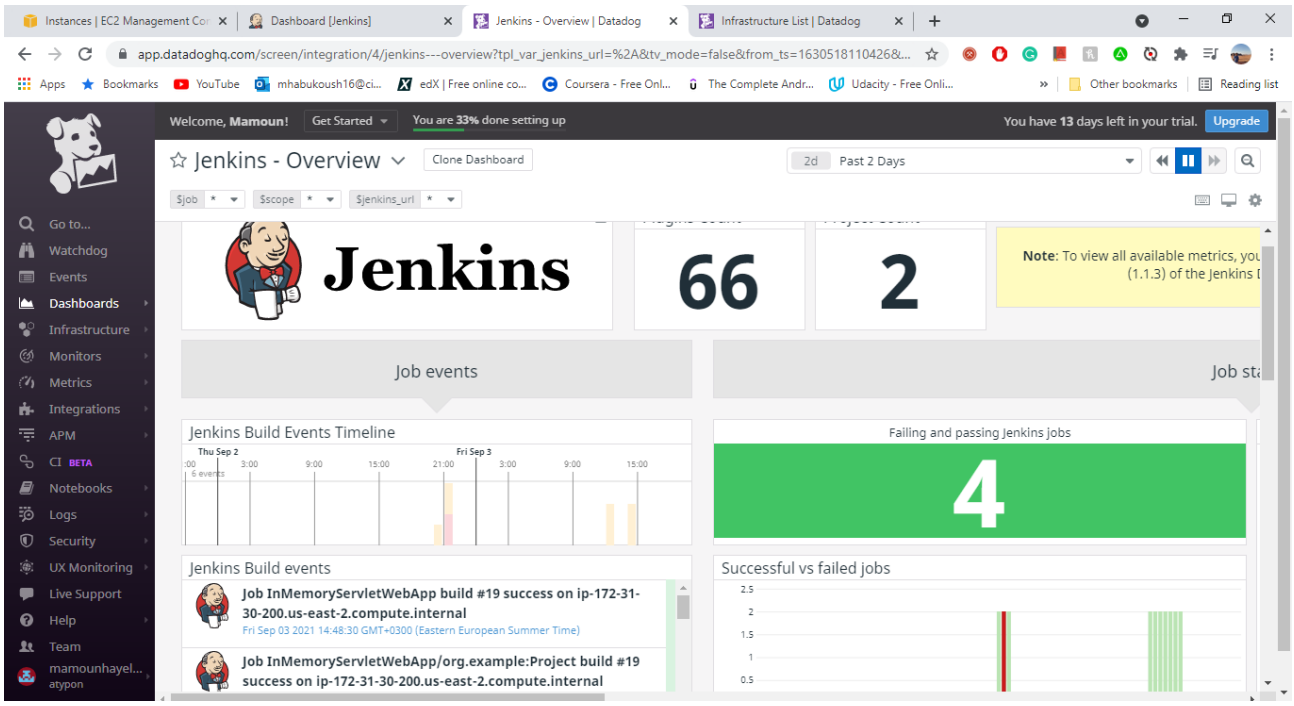


*Figure 19. Jenkins Monitor.*

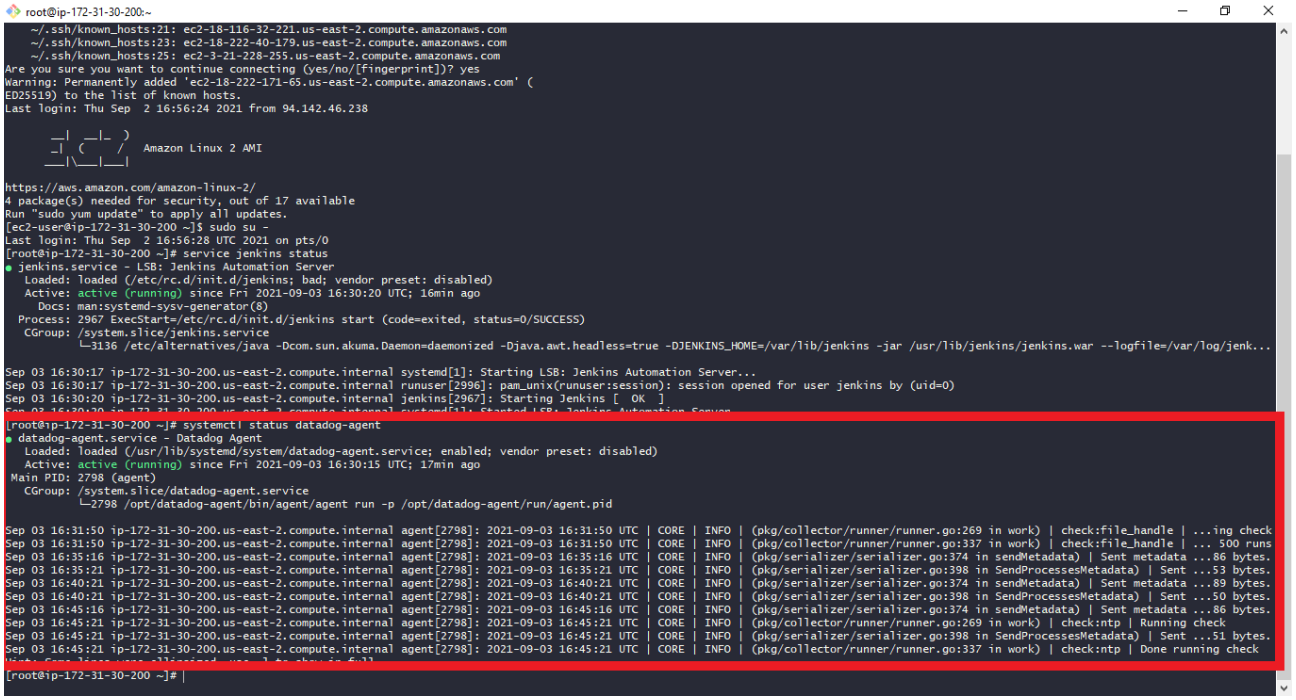


*Figure 20. Datadog Agent status on Jenkins Server.*

Also, to gain extra experience in the monitoring tools I have completed a Nagios course on O'Reilly platform, it was a rich course, I was amazed how we can link Nagios to Slack and make it notify the DevOps team in case of emergences and how we can specify the alerts or messages to a specific team or a person.

This phase is where the loop will start again, that's the beauty of DevOps, the process is continuous, there is no start point or end point for this process, its just a continuous evolution of a masterpiece. Continuity is the backbone of DevOps.

## 3.3 Database Architecture

We will only create one instance of the in-Memory database class, and we can have many tables connected to in-Memory database Class. Also, for separation of concerns and to provide extra security, the users database is not connected to the in-memory database, so in case of errors we don't compromise users' information, both users and library databases are persisted to disk in case of a shutdown.

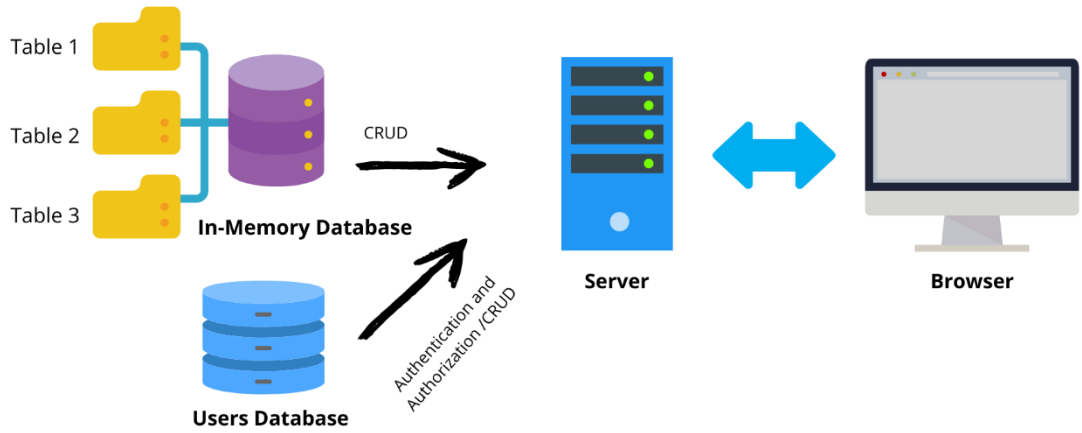


*Figure 21. Used Databases.*

For the database architecture, we are building a three-tier architecture. It's the most widely used architecture to design a DBMS. This architecture will separate and organize the application into three tiers:

1. Presentation tier (UI): this tier will be operated by the end-users and they will know nothing about the complexities or the structure behind this tier. This tier will have multiple views (HTML/JSP) of the database that will be provided by the web application, the views will be generated and managed by the application tier.

   The main purpose of this tier is to display information and collect it from the user serving as a communication layer between the user and the web application.

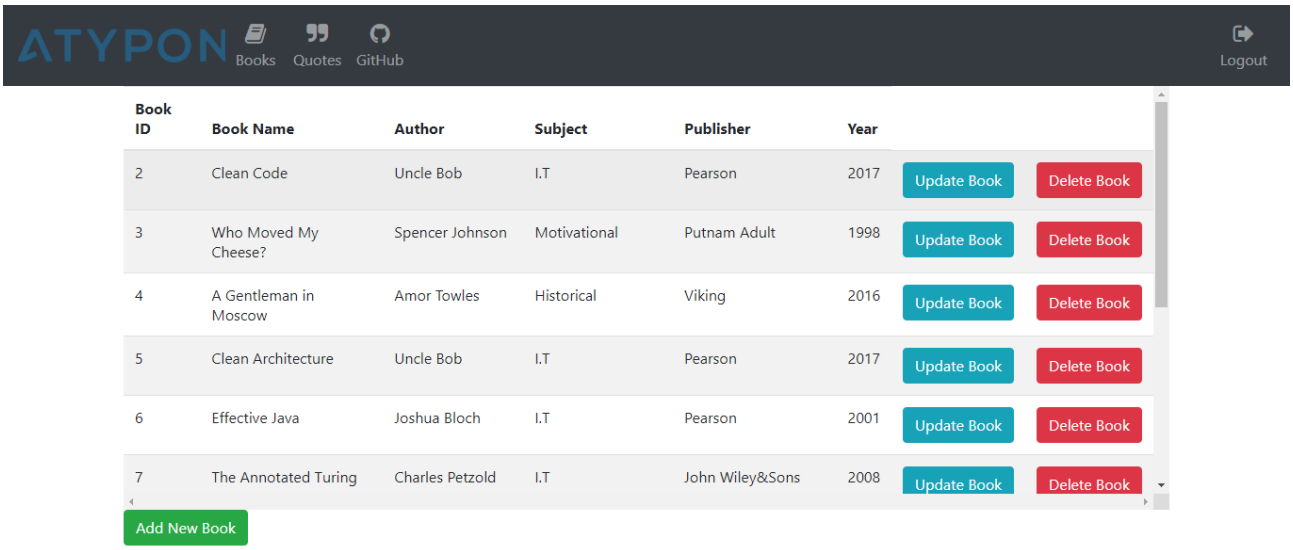


*Figure 22. Books view page (Presentation layer).*

2. Application Tier:

This tier serves as a mediator between the presentation and data tier, it's the heart of the application. In this tier all the information that is collected in the presentation tier is processed using business logic. This tier can perform All CRUD operations in the data tier.

```
@PostMapping(value = "Editor/UpdateBook")
protected String updateBook(@ModelAttribute @Valid Book book, BindingResult
result)  {

    if (result.hasErrors()){
        return "/Editor/UpdateBook";
    } else {
        inMemoryDB.bookIsUpdated(book);
        return "redirect:/Editor/Books";
    }
}
```

*Figure 23. Update book method.*

As we can see, the data will be processed in this tier, for example in above update method if the received book from the end user violates the specific set of business rules the application layer will not admit it or send it to the data tier and will inform the end user about the unsuccessful operation.

3. Data Tier

This tier is where the database resides along its processing methods and the constraints. Its where the information processed by the application is stored and managed. In our case because we are not using RDBM's we building it from scratch, the data tier is the database package with the DAO's to persist on disk.

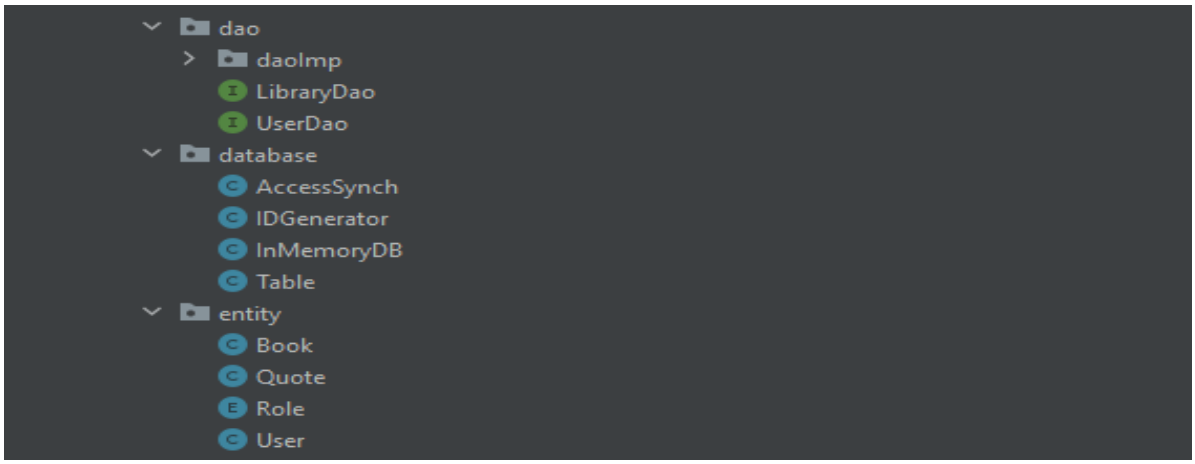


*Figure 24. Data Tier.*

- Conclusion

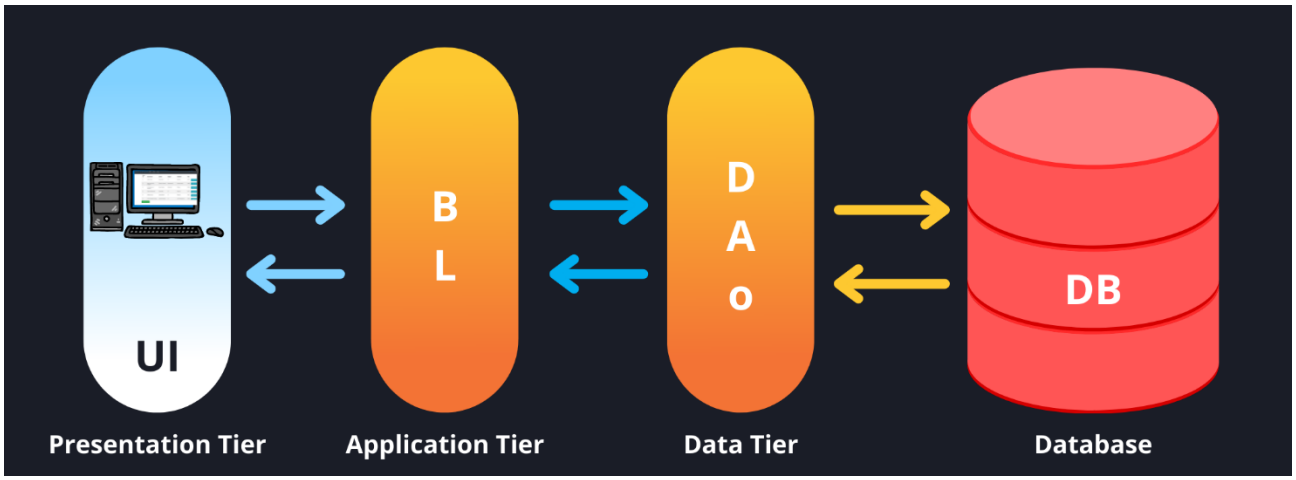


*Figure 25. 3-Tier Architecture.*

One of the main benefits of this architecture that we have a separation of functionality we can have each tier on a separate server platform.

Also, we can develop each tier simultaneously without effecting or impacting other tiers. The security will be maximised because the presentation tier and data tier cannot communicate directly because the application tier is working as a firewall preventing incorrect or malicious data to be inserted to the database.

## 1. InMemoryDB class

in this class we define the tables and their DAO's, this class will be the main central point to access the database. All tables in the library database will be initialized and accessed from this class.

```
public class InMemoryDB {

    LibraryDao<Integer,Book> bookLibraryDao = new BooksDaoImp<>();
    LibraryDao<Integer,Quote> quoteLibraryDao = new QuotesDaoImp<>();

    private final Table<Integer,Book> booksTable = new
Table<>(bookLibraryDao);
    private final Table<Integer,Quote> quotesTable = new
Table<>(quoteLibraryDao);

    private static InMemoryDB instance;}
```

*Figure 26. InMemoryDB Class.*

The InMemoryDB class will handle and manage the relations between the tables and their entities, i.e., Book with many quotes, if a book is deleted its related quotes will be deleted too but if a quote is deleted its related book won't be deleted (CASCADE), when the parent data (Book) is deleted or updated then the child data (Quote) is deleted or updated too.

## 2. ID Generator class

This class will be responsible for generating ids for both books and quotes tables records, the IDs are auto incremented and the access to the methods is synchronized to ensure that no two records get the same id.

```
protected synchronized int getNewBookId()
protected synchronized int getNewQuoteId()
```

*Figure 27. ID generator class methods.*

## 3. Table class

This class is a generic class that supports all CRUD operations, the class is flexible and accepts any object passed with the correct DAO class. When initialized it will load the table from the CSV file into the memory. We can create as many tables as we want.

```
public class Table<K,V>{

    AccessSynch<V> lock= new AccessSynch<>();

    private final TreeMap<K , V> table;
    LibraryDao<K,V> tableDao;

    private final Logger logger = Logger.getLogger("Table Logger");

    public Table(LibraryDao<K,V> tableDao) {
        this.tableDao=tableDao;
        table = tableDao.loadRecords();
    }
…}
```

*Figure 28. Table Class*

## 4. AccessSynch Class

The sole purpose of this project is to build every thing from scratch that's why I avoided using ConcurrentHashMap in the first version and in this version of the project too. AccessSynch class is used to synchronize access to the database records using Reentrant ReadWrite Locks.

```
public class AccessSynch<V>{

    private Map<V, ReentrantReadWriteLock> locks = new HashMap<>();
    private Map<V,Integer> threadQueue = new HashMap<>();
    private final Object lock = new Object();
```

*Figure 29. AccessSynch Class.*

As we can see we used Reentrant ReadWrite lock for each object in a HashMap so it can be used when multiple threads are using the database thus allowing concurrent access instead of locking the whole table.

Objects with the same id will be equal but not necessarily with the same data value, i.e., Object A and B if the have the same key (id) they are considered equal regardless to their other attributes.

The threadQueue HashMap will store the number of threads that are using or are in queue to use the record in the locks HashMap, so when there are no threads using the record, we can safely remove the record from the HashMap.

A lock object is also used to synchronize access to parts of code in the AccessSynch class that will be called only by one thread at a time. i.e., getting a record lock.

```
public void getWriteLock(final V value) throws InterruptedException {
    ReentrantReadWriteLock lock;
    synchronized (this.lock) {
        lock = obtainLock(value);
        addToThreadQueue(value);
    }
    lock.writeLock().lockInterruptibly();
}
```

*Figure 30. get write lock method in AccessSynch class.*

This method will get the write lock of the passed value, the param is final so it cannot be modified or reassigned thus ensuring the record integrity. Once the lock is obtained the thread counts will be incremented to keep track of the number of threads that are using the record.

```
public void unlockWriteLock(final V value) {
    synchronized (lock) {
        ReentrantReadWriteLock lock = obtainLock(value);
        removeFromThreadQueue(value);
        lock.writeLock().unlock();
    }
}
```

*Figure 31. unlock write lock method in AccessSynch class.*

Every get lock method call must be followed by calling the releasing method to release the held lock for a record.

The same thing goes also if we are calling the Read Lock.

```
public void update(V value, K key) {
    try {
        lock.getWriteLock(value);
        updateRecord(value, key);
    } catch (Exception e) {
        logError(e);
    } finally {
        lock.unlockWriteLock(value);
    }
}
```

*Figure 32. Update method in Table class.*

```
public V get(K key){
        try {
            lock.getReadLock(table.get(key));
            return table.get(key);
        } catch (InterruptedException e) {
            logError(e);
        } finally {
            lock.unlockReadLock(table.get(key));
        }

}
```

*Figure 33. Get method in Table class.*

Finally, this how the methods from table class would look like with the help of AccessSynch class, by using this strategy we ensured concurrent and safe access to the database, Maximizing the performance and obeying the ACID criteria.

## 3.4 Clean Code

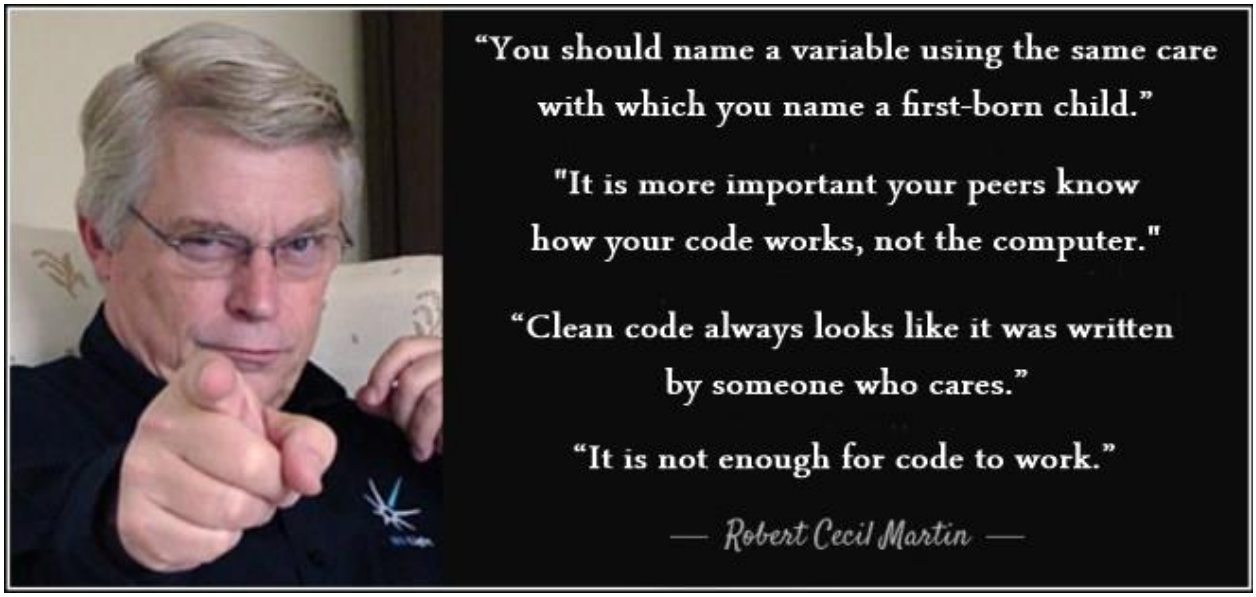


*Figure 34. Uncle Bob Quotes.*

Reading clean code book and watching uncle bob videos gave me a whole new perspective about coding, I tried my best to make sure that the code reads like a well-written prose as Grady Booch describes Clean Code. Also, I attended 2 live training sessions on O'Reilly for uncle bob, those sessions also helped me clear up some misconceptions, I wrote my notes and did my best to stick to them. Here is few of them.

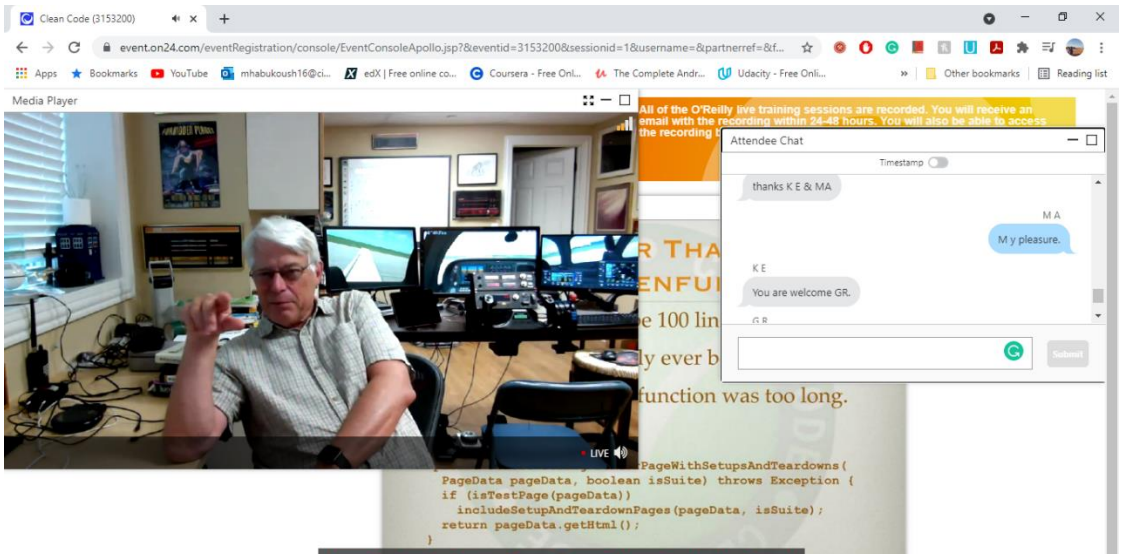


*Figure 35. Clean Code Live session on O'Reilly with uncle bob.*

# 1. Chapter 2 – Meaningful Names

- **Oettinger's variable and class naming rules:**

This paper was written back in 1997 and it is still relevant and applicable as it was written today. it helped me a lot to give my variables, methods and classes good names. By using meaningful and pronounceable variable names. Most variables names in the code are descriptive and unambiguous names.

```
public TreeMap<K, V> getAll() {…}

public V get(K key){…}

public void add(V value, K key){…}

public void update(V value, K key) {…}

public void remove(K key) {…}

private boolean isStorageFull() { return table.size() == CAPACITY; }

public boolean recordExists(K key){}

private void logError (Exception e){}
```

*Figure 36. Naming example 1.*

As we can see the names here are meaningful and descriptive, we can know what these functions do by their names.

```
ClientSession session = new ClientSession();
session.start(out,in,socket);
```

*Figure 37. Naming Example 2 - Version 1 of project.*

- **Intention-Revealing Names:**

The name of the implemented variables, methods or classes in the web project tell us why they exist and what they do and how do we use them.

- **Avoid Disinformation:**

Using names that obscure the meaning of the code was totally avoided.

- **Make Meaningful Distinctions:**

Noise words were avoided, we don't use two methods or variables with the same name by misspelling one or adding a number to the end of that variable or method i.e., A1, A2.

- **Use Pronounceable Names:**

Our goal is high code readability, so using unpronounceable names were avoided.

- **Avoid Mental Mapping:**

No code reader should have to mentally translate the variables/method names into names they know, the names in the code are clear and others can easily understand what they mean.

- **Class Names:**

The classes used have noun or noun phrase names like USER, Table, Cache.

- **Method Names:**

Implemented methods have verb names such as Add, Get, Update.

- **Don't be Cute:**

Using Cute names is not a good practice and its unprofessional.

- **One Word Per Concept:**

Instead of using many words for the same concept (Add, Insert, Put, append) we picked one word and sticked to it (Add).

# 2. Chapter 3 – Functions

- **Small – Do One Thing**:

"The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that." — Robert C. Martin.

```
public String addUser(User user, BindingResult result){

    if (result.hasErrors()){
        return "/Admin/AddUser";
    } else {
        return saveUser(user);
    }
}
```

*Figure 38. Function Example 1.*

Most functions in the implemented program are small and do one thing, one thing only, and they do it well. Flag arguments are avoided.

```
@NotNull
private String saveUser(User user) {
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    userRepo.createUser(user);
    return "redirect:/Admin/Users";
}
```

*Figure 39. Function Example 2.*

- **Function Arguments**:

I did my best to limit the amount of method arguments, because the less the function arguments the easier the testing process will be, most functions in the code have at most two arguments.

- **Flag Arguments**:

Passing a Boolean into a function is considered a bad practice. it was avoided. the functions should do only one thing with flag arguments id will do more than one thing because the behavior of the function will change if the flag is true or false.

- **Function Names**:

"You know you are working on clean doe when each routine turns out to be pretty much what you expected" – Uncle Bob.

Choosing good descriptive names will clarify the design of the code and will help to improve it, Function names say what they do.

- **Try/Catch Blocks**:

Most try/catch blocks in the program are extracted to make the code easier to understand and modify.

```
@Override
public boolean recordIsAdded(V value) {

        try(FileWriter fileWriter = new FileWriter(DBPATH, true) ) {
            addNewRecordToDB(value, fileWriter);
        } catch (Exception e) {
            logError(e);
        }
}
```

*Figure 40. Try/Catch Example.*

- **DRY Principle**:

To avoid duplicate code, instead of making new table class (Books table, Quotes table) for each new table, I created a generic table class with all CRUD operations. This way we don't have to make a new class for each new table.

```
private final Table<Integer,Book> booksTable = new Table<>(booksDao);
private final Table<Integer,Quote> quotesTable = new Table<>(quotesDao);
```

*Figure 41. Table instances.*

All we need to do to have our table ready with all CRUD operations is to pass the object type and the correct DAO so the data can be persisted on disk for changing queries.

# 3. Chapter 4 – Comments

Most comments used are legal, and to clarify what a method is doing by providing Useful information about the implementation. Misleading, Noise and nonlocal comments were avoided. i.e., comments were used here to explain the implementation of getHashedPassword() method since this method is dealing with security hashing algorithm.

```
    // Get a SHA-512 algorithm
    MessageDigest messageDigest = MessageDigest.getInstance("SHA-512");
    // add the salt to the massageDigest and digest the password
    messageDigest.update(salt);
    byte[] byteData = messageDigest.digest(password.getBytes());
    messageDigest.reset();
    return byteData; // Return the Hashed Password
}
```

*Figure 42. Comments Example.*

# 4. Chapter 5 – Formatting

- **Vertical Formatting**:

Classes used are relatively small files, none are over 300 lines and most of those files are less than 200-250 lines. Small file is considered to be very desirable as they easier to understand.

- **Variables are declared next to their usage**.
- **Most code lines are short**.
- **Dependent Functions are close, and the caller is above the Calle**.

```
@Override
public void createUser(User user) {
    configureUserPassword(user);
    saveUser(user);
}

private void configureUserPassword(User user) {...}
```

*Figure 43. Close Dependent Functions Example.*

- **Conceptual Affinity**:

Functions with strong Conceptual Affinity are close to each other. i.e., functions with similar naming scheme and similar tasks in PasswordHash Class.

```
public byte[] getSalt()  {...}

public byte[] getSaltedHash(String password, byte[] salt) {...}

private byte[] getHashedPassword(String password, byte[] salt) throws NoSuchAlgorithmException {...}

public byte[] fromHex(String hex){...}

public String toHex(byte[] array){...}
```

*Figure 44. Conceptual Affinity Example.*

- **Vertical Ordering**

Function call dependencies point in the downward direction. The classes most important methods come first, so anyone who reads the class can get the gist from the first few functions without having to go deep in the code.

```
@Override
public List<User> getAll(){…}

@Override
public User get(String email){…}

@Override
public void add(User user){…)

@Override
public void update(User user, String email){…}

@Override
public void remove(String email) {…}

@Override
public boolean userExists(String email){…}

private void logError (Exception e){…}
```

*Figure 45. Vertical ordering example.*

- **Horizontal Alignment**

Its not a good practice to use horizontal alignment as it may lead to read the variable names without looking at their type or seeing the assignment operator.

- **Indentation**

Without indentation programs would be virtually unreadable by humans, I avoided breaking indentation rules mentioned in clean code book by uncle bob.

# 5. Chapter 6 – Objects and Data Structures

- **Variables are private, only accessed by getters and setters.**

- **Avoided Train Wrecks.**

- **Minimized coupling between classes, Law of Demeter.**

- **Public interfaces:**

Interfaces are used to expose only members needed externally; no implementation details are exposed. i.e., in UserDao interface only needed methods are declared other methods that are needed internally are not exposed.

```
public interface UserDao {

      User get(String username);
      void update(User user, String email);
      List<User> getAll();
      void remove(String username);
      void save(User user) ;
      boolean userExists(String username);

    }
```

*Figure 46. Public interface example.*

# 6. Chapter 7 – Error Handling

- **Provide Context with Exceptions.**

Each exception will provide enough context and informative error messages to the user to determine the source and location of the error.

```
Sep 05, 2021 5:42:31 PM com.atypon.dbproject.dao.daoImp.BooksDaoImp logError
SEVERE: .\src\main\resources\bookDetails.csv (The process cannot access the file because it is being used by another process)
Sep 05, 2021 5:42:31 PM com.atypon.dbproject.database.Table addToTable
INFO: Record couldn't be added...
```

*Figure 47. Add new record exception.*

- **Don’t Return Null.**

For example, if a user entered wrong credentials to login, the factory design pattern will not return null value, instead it will return an invalid User type.

- **Don’t Pass Null.**

Passing null to methods is way worse than returning null values, in the web application I avoided passing null to methods.

# 7. Chapter 9 – Unit Tests

- **Clean Tests.**

The readability of tests is very important, so I made sure to keep the tests clean. Also, Without tests we won’t be sure if we broke anything after every code update.

With clean tests we can achieve very high confidence about the implemented structure.

- **One Assert per test.**

To make each test easy to understand its recommended to have only one assert per test. Also having one assert only will help if the unit test fails so we would know as precisely as possible what is the exact failure and what is going wrong with the tests.

Also having more than one assert per test is not necessarily a bad practice, because we will end up writing more tests than expected and our tests would end up testing one thing at a time.

```
@Test
public void testIsNullNormalCases(){
    Book obj = new Book.BookBuilder().ID(1).name("test").author("test")
                    .subject("test").publisher("test").year("test").build();
    assertFalse(InputExceptions.isNull(obj));
}

@Test
public void testIsNullEmptyCases(){
    assertTrue(InputExceptions.isNull(""));
}
```

*Figure 48. Input Exceptions Tests*

As we can see we are testing for two kinds of cases, I wrote the first test to confirm that a unit is behaving as expected on normal input (Happy flow or sunny-side testing). Also, in the second test method we confirm that a unit behaves reasonably when dealing with abnormal input (Unhappy flow or rainy-side testing).

- **Single Concept per Test.**

This rule is very helpful, we shouldn't test multiple concepts at the same test function. Instead to keep the test methods short we should write one test per each concept that we need to verify.

```
@Test
public void getBookTest() {}

@Test
public void getQuoteTest() {}

@Test
public void bookIsAddedTest() {}

@Test
public void quoteIsAddedTest() {}

@Test
public void bookIsUpdatedTest() {}

@Test
public void quoteIsUpdatedTest() {}
```

*Figure 49. Test methods.*

- **F.I.R.S.T.**

  - **Fast:** Implemented tests are fast and small.

  - **Independent:** Tests does not depend on each other or on the state of any previous test.

  - **Repeatable:** Tests are repeatable in any environment without change in expected results.

  - **Self-Validating:** All tests have Boolean output, either they pass or fail.

  - **Timely*:*** Unit tests should be written before the production code. As I didn't use TDD I wrote most of the tests after the production code.

# 8. Chapter 10 – Classes

- **Class Organization**

  Implemented classes members are organized in the following order:

  a. Public static constants
  b. Private static variables
  c. Private instance variables
  d. Public methods
  e. Private methods

- **Small.**

  Classes should be small and then should be smaller than that, with methods we count lines but with classes we count responsibilities, in the implemented web application we don't have any classes with too many responsibilities.

- **SRP.**

  The single responsibility principle is one of the most important principles in SOLID and the simplest to understand and adhere to. Mostly, all implemented classes have only one reason to change or one responsibility. I'll talk in detail about SRP in the SOLID section of the report.

- **Organizing for Change**

  The implemented classes should be open for extension but closed for modification (OCP). We can allow new functionality to be added via subclassing.

- **Isolation from Change**

  Here we are talking about Dependency Inversion Principle (DIP). Most implemented classes depend upon abstractions, not on concrete details.

# 9. Chapter 12 – Emergence

According to Kent Beck the creator of extreme programming development methodology, a good design should follow these 4 rules: Run all tests, contains no duplication, Express the intent of programmers and minimizes the number of classes and methods. I did my best to adhere to these rules as I was making progress and then stop for a while and refactoring the code by eliminating duplication and ensure that the code is expressive.

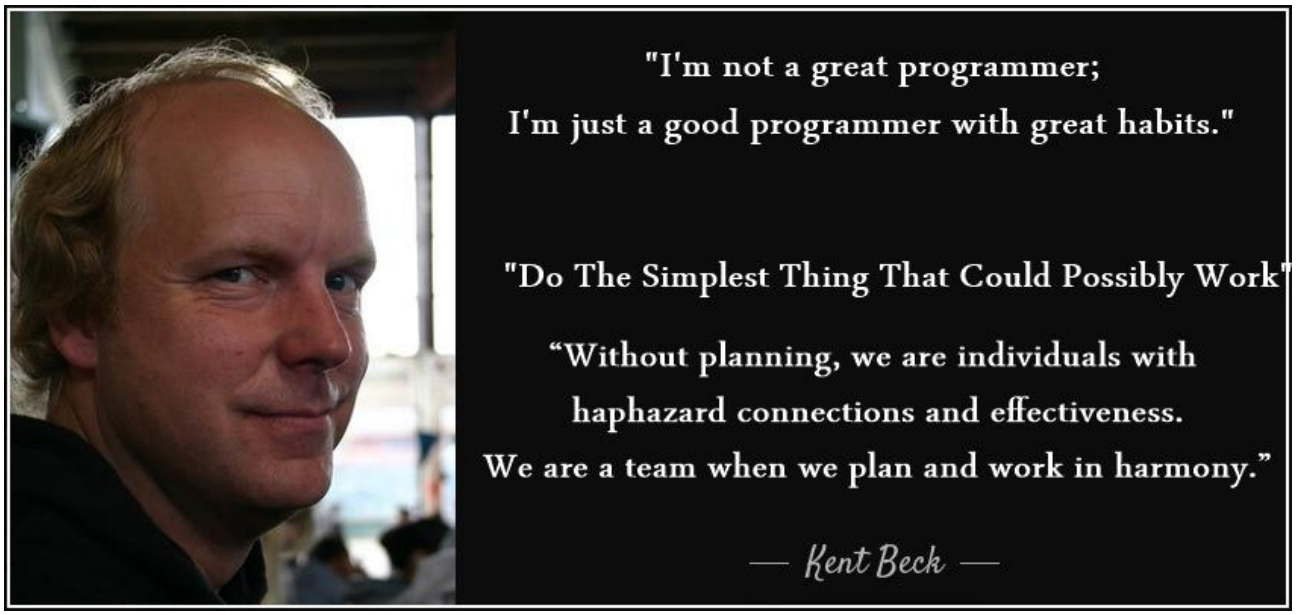


*Figure 50. Kent beck quotes.*

## 3.5 Effective Java

After reading this book, effective java is THE BOOK to read first if you want to write java code professionally. Some of the items were a little bit advanced and needed more studying, others were not related to the project. Also, I found many items in the book very USEFUL in this project.

- **Item 1:** Consider static factory methods instead of constructors:

  Static factory is used to return an instance of the inMemoryDB class, they make the code more readable because a static factory with a good name makes the code easier to read and use. And we are no longer required to create a new instance each time we invoke it.

- **Item 2:** Consider a builder when faced with many constructor parameters:

  A builder pattern was used to create objects for all entities, builder is more readable, and the arguments are now clearer. And we don't have to pass unnecessary parameters. Also, in the spring version of the project we are using @Builder annotation instead of using a hardcoded builder pattern.

```
return new User.UserBuilder().username(username).fName(fName).lName(lName)
        .password(password).role(role).build();
-------------------------------------------------------------------------------
return new Book.BookBuilder().name(name).author(author).subject(subject)
        .publisher(publisher).year(year).build();
-------------------------------------------------------------------------------
return new Quote.QuoteBuilder().bookId(bookId).quote(quote).build();
```

*Figure 51. Builder Pattern*

- **Item 3:** Enforce the singleton property with a private constructor or an Enum type:

  a private constructer was used to enforce the singleton property, one instance for all clients.

```
public class InMemoryDB {

    private static InMemoryDB instance;

    public static synchronized InMemoryDB getInstance(){
        if (instance == null) {
            instance = new InMemoryDB();
        }

        return instance;
    }
}
```

*Figure 52. In-Memory DB Class.*

- **Item 4:** Enforce noninstantiability with a private constructor:

  Since Input Exceptions Class is a collection of static methods, so it shouldn't have a constructor. So, its constructor should be private.

```
public class InputExceptions {

    private InputExceptions() {
        throw new AssertionError();
    }
}
```

*Figure 53. Input Exceptions Class.*

- **Item 5:** Prefer dependency injection to hardwiring resources:

  Using Dependency injection provides flexibility, not every class in the system needs to be singleton or non instantiable. sometimes classes behavior is parameterized by underlying resources. In these cases, we pass the resource to the class. We can do that by passing the resource through constructor when creating a new instance of the class. In the spring version dependencies are managed by spring.

```
@Controller
@RequestMapping("Admin")
@AllArgsConstructor(onConstructor = @__(@Autowired))
public class AdminController {

    private UserDao usersDao;
    private UserAccountConfig userAccountConfig;
```

*Figure 54. Using Spring to manage dependencies.*

- **Item 6:** Avoid creating unnecessary objects:

  No new objects are created when we can reuse an existing one. Common use of one method to increase code reusability.

- **Item 8:** Avoid finalizers and cleaners.

  Since using finalizers and cleaners may cause severe performance issues and security problems so using them was avoided.

- **Item 9:** Prefer try-with-resources to try-finally:

  It's the best way to close resources and its more readable, Since we are dealing with files, all methods that deal with resources uses try-with-resources, to ensure the resources are closed.

```
@Override
public Map<String, Book> loadRecords() {

    Map<String, Book> map = new LinkedHashMap <>();

    try (FileReader fileReader = new FileReader(DBPATH);
         BufferedReader bufferedReader= new BufferedReader(fileReader)) {
         loadDBIntoMap(map, bufferedReader);
    } catch (Exception e) {
        logError(e);
    }
    return map;
}
```

*Figure 55. Try-With-res.*

- **Item 10:** Obey the general contract when overriding equals:

  It was appropriate to override equals since logical equality differs from mere object identity, the overridden method adhere the general contract of equivalence relation with its 5 properties.

- **Item 11:** Always override hashCode when you override equals

- **Item 12:** Always override toString:

  All entities override the toString method, I used it as a way to map the object to the csv file as a comma separated values when its written.

```
@Override
public String toString() {
    return ID + "," + name + "," + subject + "," + author + ","
            + publisher + "," + year +"\n";
}
@Override
public String toString() {
    return id + "," + bookId + "," + quote +"\n";
}
@Override
public String toString() {
    return fName + "," + lName + "," + username + "," + password + ","
            + salt + "," + role.toString() +"\n";
}
```

*Figure 56. Entities classes toString method.*

Since I have implemented a generic DAO, the filewriter will just have to call the toString method of the object to be written to the database file without worrying about how the values should be arranged in the csv file.

```
private void addNewRecordToDB(V value, FileWriter fileWriter) throws
IOException {
    fileWriter.write(value.toString());
    fileWriter.flush();
}
```

*Figure 57. add New Record to DB method.*

- **Item 15:** Minimize the accessibility of classes and members:

  Information Hiding and Encapsulation is very important, since classes with public mutable fields are not thread-safe I Made sure that each class or member is as inaccessible as possible.

```
public class Quote {

    private final int id;
    private final int bookId;
    private final String quote;
    private final String bookName;
}
```

*Figure 58. Quote class members.*

- **Item 16:** In public classes, use accessor methods, not public fields.

  Encapsulation of data using getters/setters. Class members cannot be accessed directly, a getter should be used.

```
public class Quote {

    public int getId() {
        return id;
    }

    public int getBookId() {
        return bookId;
    }

    public String getQuote() {
        return quote;
    }
}
```

*Figure 59. Quote Class getter methods.*

- **Item 17:** Minimize mutability.

  Most Variables are final and private to provide data protection, most entity classes are immutable so they are thread-safe and require no synchronization, unless there is a good reason. Since I have used a builder pattern in entity classes I have removed setter methods to provide immutability.

- **Item 18:** Favor composition over inheritance.

  inheritance is powerful but its heavy and problematic because it violates encapsulation. To avoid many issues that is caused sometimes by inappropriate use of inheritance is to get around these issues and use composition.

```
public class Table<K,V> extends AccessSynch<V> {…}
```

*Figure 60. Table Class – Might cause issues.*

Here, the AccessSynch Class which responsible of synchronizing access to database elements values using read write locks, instead of extending that class, we create a private instance of AccessSynch class in Table class and the methods can call the methods on AccessSynch on this instance, this is known as Forwarding.

```
public class Table<K,V>  {

    AccessSynch<V> AccessSynch = new AccessSynch<>();
…
```

*Figure 61. Table Class with forwarding.*

- **Item 20:** Prefer interfaces to abstract classes:

  Interfaces gives more flexibility; a class can have one super class but it can implement so many interfaces. We can use default methods in interfaces or We can combine an interface with abstract classes (Skeletal Implementation) implementing that interface to maximize the benefits and to avoid fat interfaces with default methods as I did in the earlier version of the project.

- **Item 21:** Design interfaces for posterity:

  This item is related to item 20, default implementation of interface methods we avoided and if implemented we should deal with it very carefully, as it's mentioned in the book it is still of the utmost importance to design interfaces with great care.

- **Item 22**: Use interfaces only to define types:

  Using constant interface pattern is a poor use of interfaces, having interfaces to hold constants is wrong and should be avoided, no final values were declared in the interface.

- **Item 23:** Prefer class hierarchies to tagged classes:

  As the book clarified, tagged classes are verbose, error-prone and inefficient, so using a class with a tag filed was avoided as it's a bad practice and it will cost more memory. Also, it violates SRP.

- **Item 24:** Favor static member classes over nonstatic:

  Nested class should only exist to serve its enclosing class. Such as User class it has User Builder class inside of it and we can refer to building user operations like User.UserBuilder its like a helper class. Other example is Input Exceptions class with its static members it is useful in more contexts so it's a top-level class.

```
public class User {

    private User(UserBuilder builder){…}

    public static class UserBuilder {…)

}
```

*Figure 62. User Class.*

- **Item 25:** Limit source files to a single top-level class:

  No two classes or interfaces were defined in the same source file, defining multiple top-level classes in the same file is confusing, messy and can result in errors depending, so it was avoided.

- **Item 26:** Don’t use raw types:

  With raw types we will get compile-time exception, so using them was avoided.

- **Item 27:** Eliminate unchecked warnings

  Unchecked warnings are too important, I Eliminated every unchecked warning possible.

- **Item 28:** prefer lists to arrays

  Arrays provide runtime type safety but not compile-time safety. And arrays are deficient. And since the database is using generics, I avoided using arrays because we might get compile errors.

- **Item 29:** Favor generic types.

  All algorithm methods in Collections are generic, In the earliest version of the project, I designed it to be generic. Generic types are safer and easier to use. The table class Is generified, it made the process of creating and adding different tables easier.

```
public class InMemoryDB {

    private Table<Integer,Book> booksTable = new Table<>(bookLibraryDao);
    private Table<Integer,Quote> quotesTable = new Table<>(QuoteLibraryDao);
…
```

*Figure 63. InMemoryDB Class.*

- **Item 30:** Favor generic methods.

  The same thing as classes, methods can be generified and we have the same benefits as generic types they are safer and easier to use than methods that requires specific cast on the passed parameters. Most methods used in the code are generified.

```
public TreeMap<K, V> getAll() {
    return table;
}


public void update(V value, K key) {
    try {
        lock.getWriteLock(value);
        updateRecord(value, key);
    } catch (Exception e) {
        logError(e);
    } finally {
        lock.unlockWriteLock(value);
    }
}
```

*Figure 64. get All and update generic methods.*

- **Item 34**: Use Enums instead of int constants:

  Since user types ae a fixed set of constants we should use enum otherwise the alternative is constant strings or ints which is a bad practice.

```
public enum Role {
    ADMIN, EDITOR, VIEWER
}
```

*Figure 65. Role enum.*

- **Item 40:** Consistently use the Override annotation:

  It's a good practice to use @Override annotation on all methods that override superclass or interface methods, the @Override annotation was used on method declarations that override declarations from interfaces as well as classes.

```
@Override
public TreeMap<K, V> loadRecords() {…}

@Override
public boolean recordIsAdded(V value) {…}

@Override
public boolean recordIsDeleted(K key){…}

@Override
public boolean recordIsUpdated(V value, K key)   {…}
```

*Figure 66. Books Dao class methods.*

- **Item 42:** Prefer lambdas to anonymous classes:

  Lambda functions are just like mathematical functions and they ease functional programming. I made sure to use Only one line in a lambda expression because they are not self-explanatory and may result in confusion sometime and make the code hard to read.

```
quotesTable.getAll().entrySet().removeIf(entry ->
entry.getValue().getBookId() == key);
```

*Figure 67. Lambda Expression.*

- **Item 49**: Check parameters for validity:

  Since it's a database system, we must adhere to the acid criteria so we need to check every value for validity, that's why I implemented **InputExceptions** class to check parameters for validity.

```
public static boolean isNull(final Object reference) {
    String checker = reference.toString();
    String[] newStrings = checker.split(",");
    for (String newString : newStrings) {
        if (isEmpty(newString)) {
            return true;
        }
    }
    return false;
}
```

*Figure 68. is Null method.*

In the above isNull method it will check every object whether it's a book or a quote or a user entity before adding it to the database if it has a null value it will not be added and the user will be notified.

Also, the same thing is applied in the spring version of the project using the validation constraints annotations.

No new object can be added to the database if it does not fulfil the required constraints.

```
@NotEmpty(message = "* Please provide user Email")
@Email(message = "* Please provide a valid Email")
String email;

@NotEmpty(message = "* Please provide user First Name")
String fName;

@NotEmpty(message = "* Please provide user Last Name")
String lName;

@Length(min = 8, message = "* User password must have at least 8 characters")
@NotEmpty(message = "* Please provide user password")
String password;

@NotEmpty(message = "* user cannot be created without an assigned role")
String role;
```

*Figure 69. User Class.*

- **Item 50**: Make defensive copies when needed:

  Methods were designed with the assumption that the client will do their best to destroy it. So, we predicted how things can go wrong and how we can prevent them.

- **Item 51:** Design method signatures carefully:

  Obeyed clean code-naming rules and Oettinger's paper, methods and variables have the best names that clarify their purpose, long names were avoided. Implementing Methods with long parameters list were avoided.

- **Item 52**: Use overloading judiciously:

  Using the same name for two or more methods but with different signatures such as (AddRecord) for both books and quotes is very confusing and hinder the code readability. In the below situation, I used different names instead of 2 overloaded methods.

```
public boolean bookIsAdded(Book book){…}
public boolean quoteIsAdded(Quote quote){}
```

*Figure 70. InMemoryDB methods.*

- **Item 54**: Return empty collections or arrays, not nulls:

Returning null sometimes make the code messy, no methods used return null values, instead they return empty maps.

- **Item 57:** Minimize the scope of local variables:

  Variables are declared next to their usage, also this item is related to clean code practices such is keeping methods small a focused, I made sure not to implement methods that do more than one thing, methods do one thing, one thing only, and they do it well.

- **Item 58:** Prefer for-each loop to traditional for loops:

  For-each loop is clearly expressed, less error prone and more flexible, and they come without any performance penalties, so I have implemented for-each loop instead of traditional loop.

```
private void assignBookNames() {
    Set<Integer> keys = quotesTable.getAll().keySet();
    for(Integer k:keys) {
        String bookName= booksTable.get(quotesTable.get(k).getBookId()).get-
Name();

        quotesTable.getAll().get(k).setBookName(bookName);
    }
}
```

*Figure 71. Assign Book Names method.*

As we can see in the above method, I have used enhanced for loop and also the method complies with item 57 where variables are declared next to their usage.

- **Item 59:** Know and use the libraries:

  Using libraries helped a lot, they provided many advantages since they were tested and wrote by experts but since depending on them may impose restrictions sometimes so I made sure to balance things out and not to depend fully on libraries.

- **Item 60:** Avoid float and double if exact answers are required:

  Using float and double for exact answers is not recommended because they are ill-suited for monetary calculations, The database tables doesn't have values that require engineering or scientific calculations. But if it does, I'd use Big Decimal, int or long as this item recommended.

- **Item 61:** Prefer primitive types to boxed primitives:

  Primitive types are more time and space efficient. And using boxed primitives sometimes such as (Integer, Double, Boolean) might cause issues such as comparing because it compares their identities rather than their real values. That's why I preferred using primitives such as (int, boolean).

- **Item 62:** Avoid strings where other types are more appropriate.

  as its mentioned in this item, strings do a fine job in representing text, but they are poor substitutes for other value type such as enum types that's why Role is defined by an enum instead of a string.

```
public class User {

    String username;
    String firstName;
    String lastName;
    String password;
    Role role;
}
```

*Figure 72. User Class using enum instead of a string.*

- **Item 63:** Beware the performance of string concatenation:

  Strings are immutable thus using string concatenation operator on n strings requires time quadratic in n. string concatenation Is not widely used in the code, and its only used to concatenate only few strings usually two items only.

- **Item 64:** Refer to objects by their interfaces.

  We should favour the use of interfaces rather than classes as parameter types, so if we want to switch implementations all we have to do is to change the class name in the constructor, adhering to this item makes the program much more flexible.

```
Map<K, V> booksMap = new TreeMap<>();
```

*Figure 73. referring to books TreeMap by map interface in BooksDao getAll method.*

- **Item 65:** Prefer interfaces to reflection:

  In the earlier version of the project, I was using reflections to populate the user menu and user operations factories it was very helpful since we don't know what is the menu/operation class to be used until run time. But in the web version of the project reflections was not needed and was avoided because it might affect the performance of the program and we lose the advantage of compile time checking.

- **Item 66:** Use native methods judiciously:

  Java have changed a lot and more features and libraries were added to it. So, it became more and more independent on these native methods written in native programming languages such as C/C++, in the program to improve performance we rarely use native methods.

- **Item 67:** Optimize judiciously:

  This item reminds me of the famous proverb "if it ain't broke, don't fix it" because sometime when we try to optimize code the results are usually negligible and sometimes might cause malfunction in other parts of code as Uncle bob mentioned in his video lectures. Instead, we should focus on having a good architecture, which is flexile, scalable and fast.

- **Item 68:** Adhere to generally accepted naming conventions:

  Naming conventions and clean code naming practices were applied for packages, classes, interfaces, methods and fields. No bad or cute names were chosen, a method name describes what the method is doing and so on for other practices.

- **Item 69:** Use exceptions only for exceptional conditions.

  Exceptions are used wisely, they are not abused or over used. Also, they are not used for normal control flow.

- **Item 74:** Document all exceptions thrown by each method.

  it's a good practice to document exceptions and log them, because they are usually a programmer error and by documenting them, the programmer will know what to expect and how to properly use that method or in interface.

```
Sep 05, 2021 5:42:31 PM com.atypon.dbproject.dao.daoImp.BooksDaoImp logError
SEVERE: .\src\main\resources\bookDetails.csv (The process cannot access the file because it is being used by another process)
Sep 05, 2021 5:42:31 PM com.atypon.dbproject.database.Table addToTable
INFO: Record couldn't be added...
```

*Figure 74. Documenting Exception thrown by a method.*

- **Item 75:** Include failure-capture information in detail messages:

  As we have seen in figure 70, a detailed massage of the exception was shown containing the issue that contributed to the exception, also no sensitive data is shown in exception messages such as passwords or encryption keys.

- **Item 76:** Strive for failure atomicity:

  To achieve failure atomicity, I have changed the order of the computations, so the part that may cause an exception will come first.

```
private void addNewRecord(V value, K key) {
    if (tableDao.recordIsAdded(value)){
        table.put(key, value);
        logger.log(Level.INFO,"Successfully Added new Record to the In-
Memory");}
    else {
        logger.log(Level.INFO,"Record couldn't be added... ");
    }
}
```

*Figure 75. Adding new record method from Table class.*

In the above method from table class, if a new record is added it have to be first persisted to disk, then be added to the In-Memory database so if there were any exceptions thrown during adding the new record the In-Memory database will not be effected and the method will leave it in the state it was in prior to the add operation.

- **Item 77:** Don't ignore exceptions:

  Using empty catch block is a bad practice and defeats the purpose of exceptions. Proper actions are taken to address the thrown exception. No empty catch blocks were used in the program.

- **Item 78:** Synchronize access to shared mutable data:

  Access to mutable table data is synchronized by using the help of AccessSynch class to guarantees that no method will ever observe the object in an inconsistent state or two threads editing the object at the same time.

```
public class AccessSynch<V>{

    private Map<V, ReentrantReadWriteLock> locks = new HashMap<>();
    private Map<V,Integer> threadQueue = new HashMap<>();
    private final Object lock = new Object();
…
…
…
…
}
```

*Figure 76. AccessSynch class.*

- **Item 78:** Avoid excessive synchronization:

  We should avoid excessive synchronization, not every thing should be synchronized as we may get unwanted exceptions or deadlocks and we don't want that to happen.

```
public void getReadLock(final V value) throws InterruptedException {
    ReentrantReadWriteLock lock;
    synchronized (this.lock) {
        lock = obtainLock(value);
        addToThreadQueue(value);
    }
    lock.readLock().lockInterruptibly();
}
```

*Figure 77. get Read Lock method in AccessSynch class.*

  As we can see in the above method from AccessSynch class there is as little work as possible inside synchronized region. Also, to avoid failure no control was ceded to the client within the synchronize block.

- **Item 82:** Document thread safety

  The most important classes are the InMemoryDB class and the table class because we need to protect the database integrity so I made sure that the classes are thread safe as most classes are to provide concurrent access to the database. Also, the lock object in the AcessSynch is final to make sure to prevent changing its content which might result in a catastrophic unsynchronized access.

```
private final Object lock = new Object();
```

*Figure 78. Lock Object from AccessSynch class*

- **Item 83:** Use lazy initialization judiciously

  This item as many other items relates to clean code practices, lazy initialization is used carefully, in AccessSynch class we use synchronized accessors for lazy initialization as it's the simplest and clearest alternative.

## 3.6 ACID Criteria

In this section ill describe the ACID transaction management that I implemented to insure the data validity despite errors, power cuts and other failures.

### 1. Atomicity

Since we are dealing with an In-Memory DB with persistence layer, if an add/delete/update operation failed in the database file on the HDD it will not be added to the In-Memory DB. Which means either all successful or none.

```
private void removeRecord(K key) {
    if (tableDao.recordIsDeleted(key)){
        table.remove(key);
    } else {
        logger.log(Level.INFO , "Couldn't delete record....");
    }
}
```

*Figure 79. Atomic Operation.*

In the above method in Table Class, first it will check if the record was deleted successfully from the database fille, if not then the record will not be deleted from the InMemoryDB. See bellow example in case of a failed remove.

```
Jul 26, 2021 3:38:56 PM com.atypon.dbproject.books.BooksDaoImp deleteRecordFromDB
INFO: Record Found, Deleting Record....
Jul 26, 2021 3:38:56 PM com.atypon.dbproject.books.BooksDaoImp logError
SEVERE: .\src\main\resources\bookDetails.csv: The process cannot access the file because it is being used by another process
Jul 26, 2021 3:38:56 PM com.atypon.dbproject.database.InMemoryDatabase removeRecord
INFO: Couldn't delete record....
```

*Figure 80. Server-Side Failed Delete due to file is being used by another process.*

In the above scenario the database file (CSV file) was being used by other resources so the server logged the error and the new record was not removed from the InMemoryDB and the user was notified of the failed operation.

### 2. Consistency

Many methods and a helper class were implemented to ensure that the database is consistent and correct thus we can ensure that each transaction can only bring the database from one valid state to another.

For example, if an editor user wants to add a record with an existing ID or an Admin user wanted to add a new user with valid user credentials but with invalid user role in this case Consistent rules were enforced to ensure the accuracy of the database transaction.


*Figure 81. Adding an Existing Record. (Old Version)*

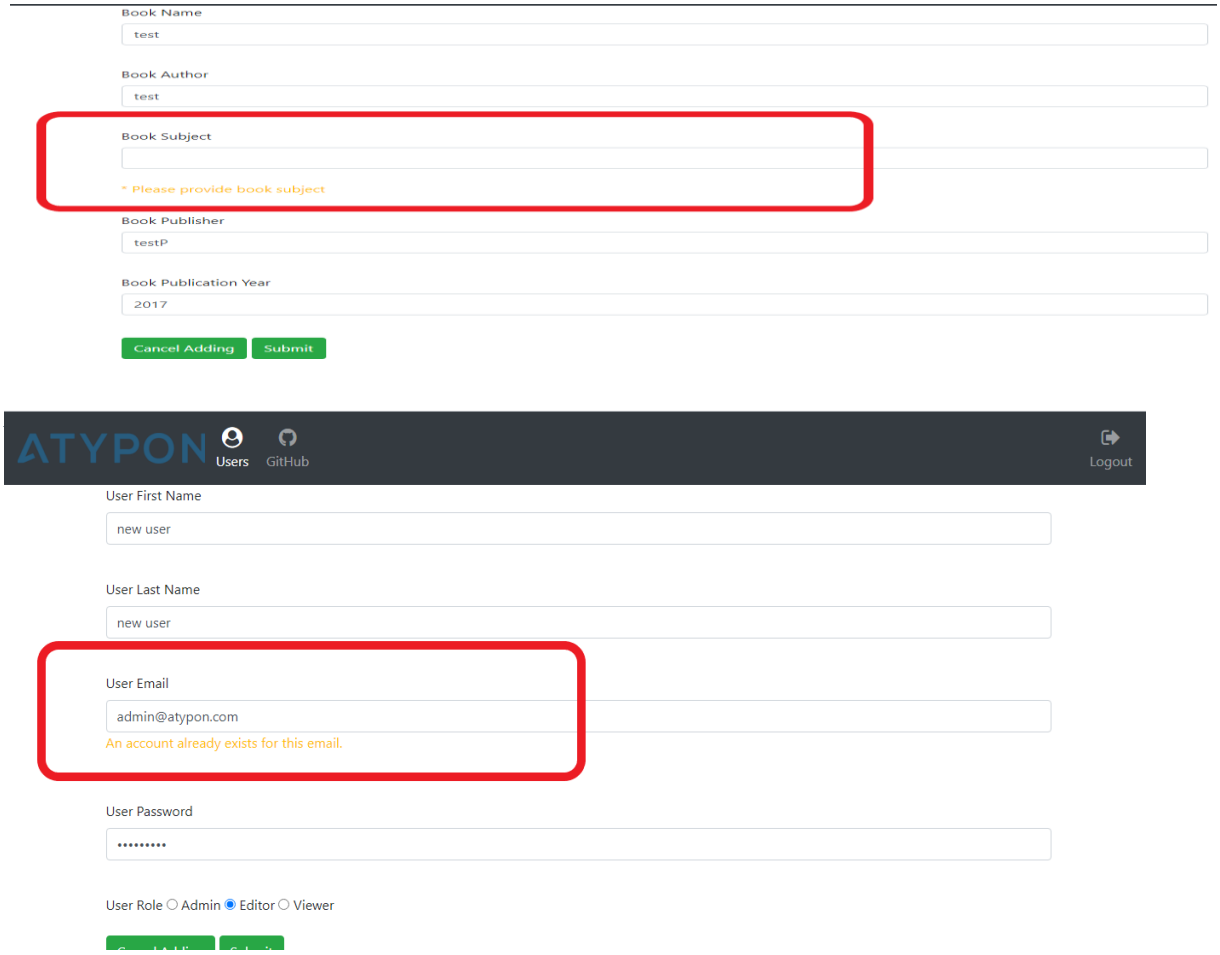


*Figure 83. Adding a user with an existing email.*

As we can see, with the help of the helper class InputExceptions and the methods implemented to check the input we can ensure that no illegal transactions, although that does not guarantee that a transaction is correct (i.e., typo in record name) but it guarantees that the transaction doesn't break the rules that were defined.

## 3. Isolation

Most transactions are executed by multiple threads at the same time concurrently, we could have multiple users preforming reading and writing operations to the database at the same time. With isolation we can ensure that each transaction is isolated from the other transactions, also any incomplete transaction will not affect the other transactions.

If we have 2 users who wants to preform transactions at the same time, both transactions will operate in an isolated manner, the database will perform a user transaction before executing the other, that will prevent producing data that is already in the process of being modified.

That's why I implemented Reentrant ReadWrite Locks for each object, since we need exclusive writes on the DB and to protect the block of code that is going to modify the InMemoryDB but also, we want to allow parallel reads of InMemoryDB. Thus, I thought using ReadWriteLock here is the right choice. Both the read lock or write lock are instances of the Lock interface.

Some rules about ReadWriteLocks:

- As many threads as we need can hold the read lock for **one object**.
- Only one thread can hold the write lock for **one object**.
- When a thread holds the write lock, no one can hold the read lock.

ReentrantReadWriteLock is more appropriate here since it offers non-block-structured locks, an option for fairness, interruptible lock waits, lock polling. Also, since read operations can be made in parallel. It thus allows for superior throughput, especially when we have more reads and few writes' operations.

```
private HashMap<V, ReentrantReadWriteLock> locks = new HashMap<>();
private HashMap<V,Integer> threadQueue = new HashMap<>();
private final Object lock = new Object();
```

*Figure 84. ReentrantReadWrite Lock.*

If multiple users requested to read a specific value from the InMemoryDB at the same time they will all acquire the read lock, and can read the value in parallel, in this case we will reduce the thread starvation possibility.

Specifically, if more viewers users were connected to the server and reading from the DB. Using ReadWriteLock will eliminate concurrency errors which usually happens when reads and writes operations occur to a shared resource concurrently, or if multiple writes take place at the same time.

```
public void update(V value, K key) {
    try {
        lock.getWriteLock(value);
        updateRecord(value, key);
    } catch (Exception e) {
        logError(e);
    } finally {
        lock.unlockWriteLock(value);
    }
}
```

*Figure 85. Update Method.*

If ReentrantReadWriteLock wasn't used and a thread is modifying the inMemoryDB, the other thread reading the inMemoryDB can get incorrect value because writing a field is not an atomic operation. In this case using

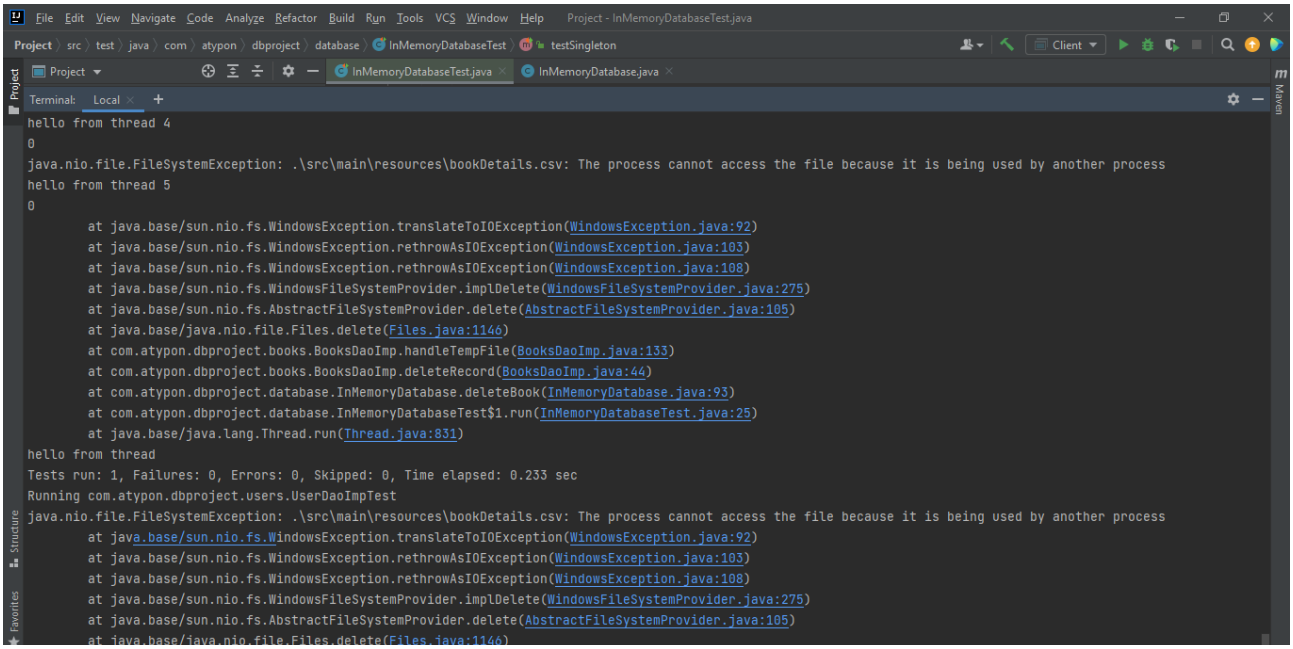


*Figure 86. Multiple Threads Accessing the DB at the same time.*

ReentrantReadWriteLock is required to maintain data integrity. As we can see in the below example without ReentrntrantReadWriteLock:

**4. Durability**

each transaction committed, will remain so, even in the event of errors or power loss. we ensured that by using a transaction log that facilitate the restoration of committed transaction despite any failure. No transactions committed to the inMemoryDB will be lost as we are using a transaction log (BookDetais.CSV).

after each write/delete/update operation the data will be saved into non-volatile storage as a CSV file. I stored the database as a .CSV file for the following reasons:

- CSV files are easier to import into a spreadsheet or another storage database regardless of the software we are using.
- CSV files are widely used to better organize large amounts of data.
- CSV format is supported by libraries available form many programming languages.
- Also, I worked with CSV files in Machine Learning and information retrieval.
- The resulting data is human-readable and can be easily viewed with a notepad or Microsoft Excel or any spreadsheet program.

## 3.7 Design Patterns

Design patterns made the design more flexible, more resilient to change, and easier to maintain. In this subsection ill talk about the design patterns I have implemented and why I chose them.

- **Singleton Design Pattern**

The In-Memory database class is a singleton, it was very useful here to use singleton design pattern since we need only one instance of the InMemoryDB to coordinate actions across the system. By using Singleton Design Pattern, we will ensure that only one instance of the DB exists.

```
public static synchronized InMemoryDatabase getInstance(){
    if (instance == null) {
        instance = new InMemoryDatabase();
    }

    return instance;
}
```

*Figure 88. InMemoryDatabase Class.*

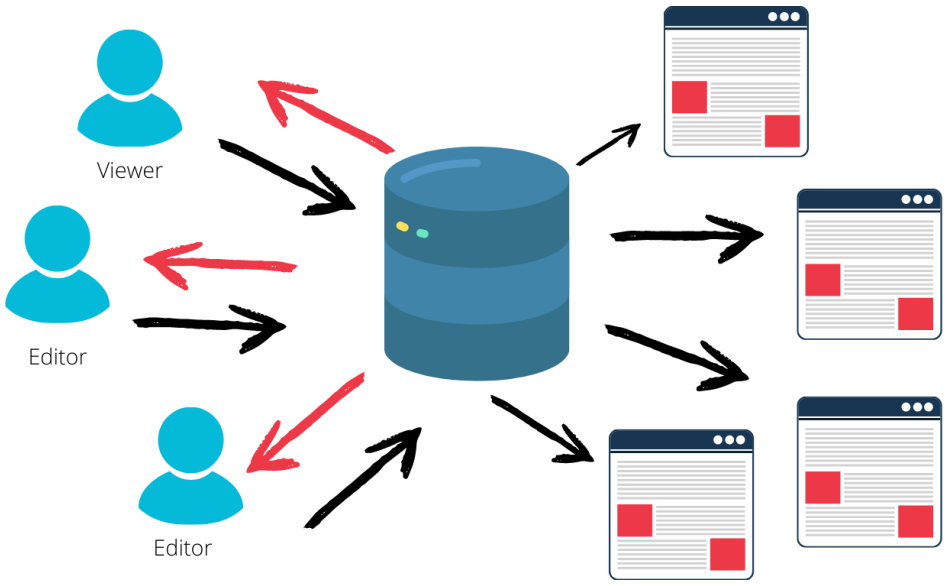


*Figure 87. in Memory Database diagram.*

- **Data Access Object Design Pattern**

To isolate the application layer from the persistence layer I implemented the Data Access Object (DAO) design pattern, it will hide all the complexities involved in preforming CRUD operations in the underlying storage mechanism. DAO is implemented for both Books and Users Database.

```
public interface BooksDao {

    void addRecord(Book book);
    void deleteRecord(String bookNum);
    void updateRecord(Book book, String bookID);
    Map<String, Book> loadRecords();

}
```

*Figure 89. Books DAO.*

```
public interface UserDao {

    User verifyCredentials(String login , String password);
    void deleteUser(String username);
    void createUser(User user) ;
    boolean userExists(String username);

}
```

*Figure 90. Users DAO.*

- **Façade Design Pattern**

Façade pattern helps to hide the complexities of the system and provide an interface to the client using the system. A façade class is implemented with simplified method to get and manage the client authentication.

```
@Component
public class AuthenticationFacade {

    public Authentication getAuthentication() {
        return SecurityContextHolder.getContext().getAuthentication();
    }

    public UserPrincipal getPrincipalUser() {
        return (UserPrincipal)
        SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    }

}
```

*Figure 91. Authentication Facade Class.*

- **Builder Pattern**

Having multiple constructors with several parameters that can grow as far as the number of the class members is called Telescoping Constructor. It was totally avoided because the class can grow and have various constructors with many parameters, with an enormous number of constructors it will be difficult to remember the required order of the parameters and it will add complexity to the code.

To solve the Telescoping Constructor anti-pattern problem, we are using Builder pattern. It makes the code more readable and reusable, its very helpful when we have many different possibilities or parameters for the constructor.

```
private Book(BookBuilder builder){
    this.ID=builder.ID;
    this.name=builder.name;
    this.author=builder.author;
    this.subject=builder.subject;
    this.publisher=builder.publisher;
    this.year=builder.year;
}
```

*Figure 92. Builder pattern.*

- **Factory Design Pattern (Older Project)**

Since we have different users with different roles and privileges, then using a factory design pattern was the best viable solution, 2 factories are implemented the first factory will retrieve the proper user menu to the client. And the other one will retrieve the proper user operations based on user role. Its more like Front-End & Back-End. Each factory deals with a side. Each factory will handle the user based on his role and privileges.

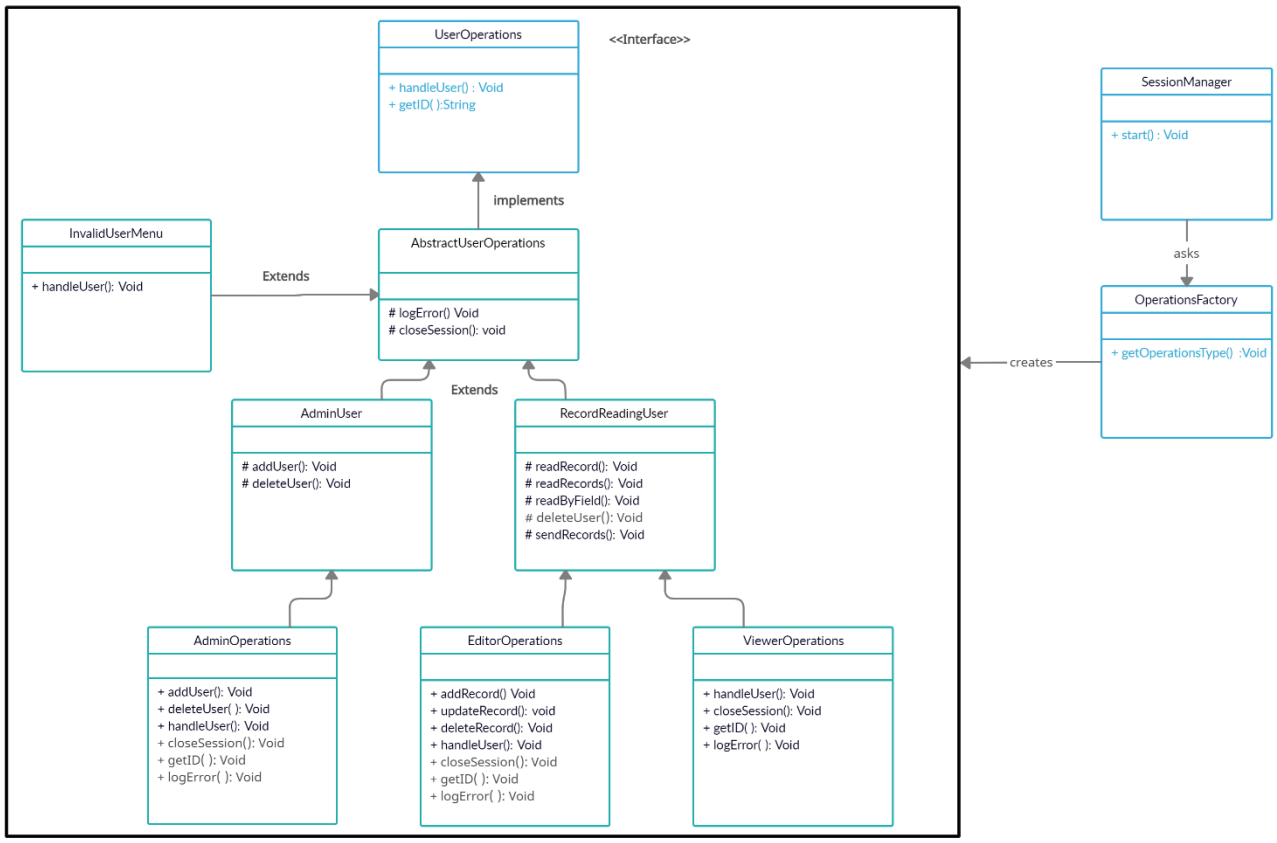


*Figure 93. User Operations Factory.*

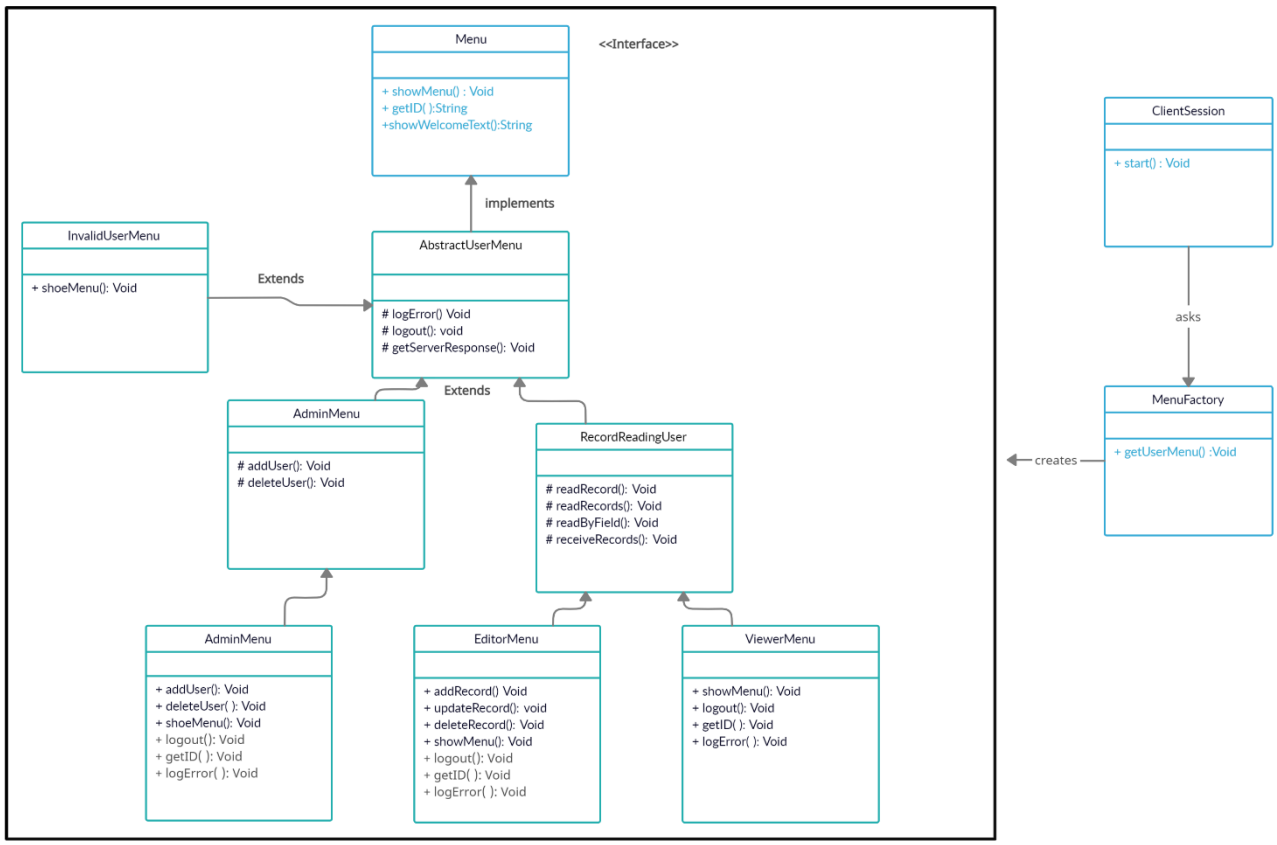


*Figure 94. User Menus Factory.*

## 3.8 SOLID

The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. These principles establish practices that lead to developing software with considerations for maintaining and extending as the project grows. They also contribute to avoiding code smells, refactoring code, and Agile or Adaptive software development.

1. Single Responsibility Principle (SRP):

   Mostly, all implemented Classes have one and only one reason to change, meaning that a class have only one job to do. For example, InMemoryDatabase class have one responsibility only which is to maintain in-memory database with CRUD operations, DB persistence is another responsibility for another class. Also, LoginService class have one responsibility, which is sending user credentials to be verified. And many more classes with only one responsibility. Following this principle made the classes easier to understand, maintain and more reusable. And more cohesive.

2. Open-Closed Principle (OCP):

   "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.".

   If we want to add a new table entity, we can do that without modifying any existing code, all we need is creating a new instance of the table class. Also, in the earlier version of the project If we visited the Operations/Menus Factory classes we would find that we used DI instead of a switch statement to fetch the right Operation or menu based on the user role.

```
public class OperationsFactory {

    Map<String, UserOperations> ops;

    public OperationsFactory(Map<String, UserOperations> ops) { this.ops = ops; }


    public UserOperations getOperationsType(User.Role role){

        return ops.get(role.toString());
    }
```

*Figure 95. Operations Factory Class*

In the above example from operations factory class the map will be loaded automatically with all classes implementing the UserOperations Interface, this way if we added a new user type. we will not violate OCP and modify any existing code, we would just add the user operations class and user menu class.

This way the Operations/Menus factories are extendable, by configuring the map with dependency injection we will not violate Open Closed Principle whenever we add a new role.

3. Liskov Substitution Principle (LSP):

   Objects are replaceable with their subtypes without affecting the correctness of the program, so if we have a parent class and a child class, both of them can be used interchangeably without getting incorrect results.

4. Interface Segregation Principle (ISP):

"Clients should not be forced to depend upon interfaces/methods that they do not use.".

Actually, at first, I was violating this principle by providing User Menus/Operations interfaces with default methods, also I was using the default method as methods for the implementing classes to call not the users of the interface. Also, the interfaces were fat. The below code would work fine in the earlier version of the program:

```
InvalidUserOperations invalidUser = new InvalidUserOperations();
invalidUser().readRecords();
```

*Figure 96. ISP Violation.*

I was violating ISP by using this approach, I fixed it using abstract classes, as the bellow code shows:

```
public interface UserOperations {

    void handleUser();
    String getID();
}
```

*Figure 97. User Operations Interface.*

All methods declared here are to be called by code that holds instances of UserOperations interface, no default methods are added.

```
abstract class AbstractUserOperations implements UserOperations {

    private final Logger logger = Logger.getLogger("UserOps");

    protected void closeSession(Socket socket){
        try {
            System.out.println("Client Left");
            socket.close();
        }
        catch (Exception e){
           logError(e);
        }
    }

    protected void logError (Exception e){
        logger.log(Level.SEVERE,e.getMessage());
    }

}
```

*Figure 98. Abstract User Operations Class.*

In this abstract class only methods and fields that are useful and valuable to the implementations of UserOperations will be added here. For example all users use closeSession() method to close the session when the user leave.

```
abstract class RecordReadingUser extends AbstractUserOperations {

    protected void readRecords(){...}
    protected void readRecordsByField(){...}

         /* etc…… */
}
```

*Figure 99. Record Reading User Class*

In Record Reading User abstract class only methods and fields that are useful and valuable to record reading users are added here (i.e., Editors and viewers).

```
final class EditorOperations extends RecordReadingUser {


    @Override
    public void handleUser(){...}

    @Override
    public String getID(){...}

    private void handleAddRequest()  {
        try {
            addNewRecord();
        } catch (Exception e) {
            logError(e);
        }

        // etc.....
}
```

*Figure 100. Editor Operations Class*

As we can see in the code above, we have achieved ISP, no classes now are forced to depend or implement methods they do not use. We have no fat interfaces; no default methods are being called by the implementing class. we could also use another alternative to achieve ISP and DRY by using a composition instead of abstract classes.

```
final class RecordReader{
 // with record reading methods
}


final class EditorOperations extends AbstractUserOperationss{

    private final RecordReadr reader= new RecordReader(){...}

    @Override
    void handleUser()  {
          reader.readRecords();

        // etc.....

         }

        // etc.....
}
```

*Figure 101. Composition instead of abstract classes.*

but I preferred using abstract class because if we wanted to add a new user role in the future i.e., a Premium User with both admin and normal user privileges all we need is a PremeuimUser final class that extends both RecordReadingUser and AdminUser abstract class. That way we can add new users without violating ISP or OCP.

5. Dependency Inversion Principle (DIP):

   At first this principle can be a little bit hare to understand, its similar to Dependency injection but they are not identical, as DIP keeps high level modules from knowing the details of its low-Level Modules, making sure the code is abiding the dependency inversion principle will reduce the coupling between classes and modules. As this is very beneficial as coupling will make the code harder to refactor.

   This principle states two important practices:

   a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
   b. Abstractions should not depend upon details. Details should depend on abstractions.

```
public class UserDaoImp implements UserDao {

    IPasswordHash passwordHash;

    public UserDaoImp(IPasswordHash passwordHash){
        this.passwordHash=passwordHash;
    }

}
```

*Figure 102. DIP Example.*

To avoid violating DIP, class PasswordHash which is not depending on any other class is implementing IPasswordHash (Absrtraction), this way UserDaoImp is no more dependent of PasswordHash its dependent on IPasswordHash interface which abiding the dependency inversion principle.

this way if we want to create another Password hashing class with different hashing algorithm, we can simply inject it into UserDaoImp using the constructor without worrying of any problems because UserDaoImp is not depending on concrete objects its depending on abstractions (IPasswordHash interface).

## 3.9 Access Control and Security

In this section, I'll explain how I implemented Role-based access control and the security measures in both versions of project (Spring and Traditional Servlets/Jsp) to maximize the system security and how the users accounts are created, checked, stored in the user's database.

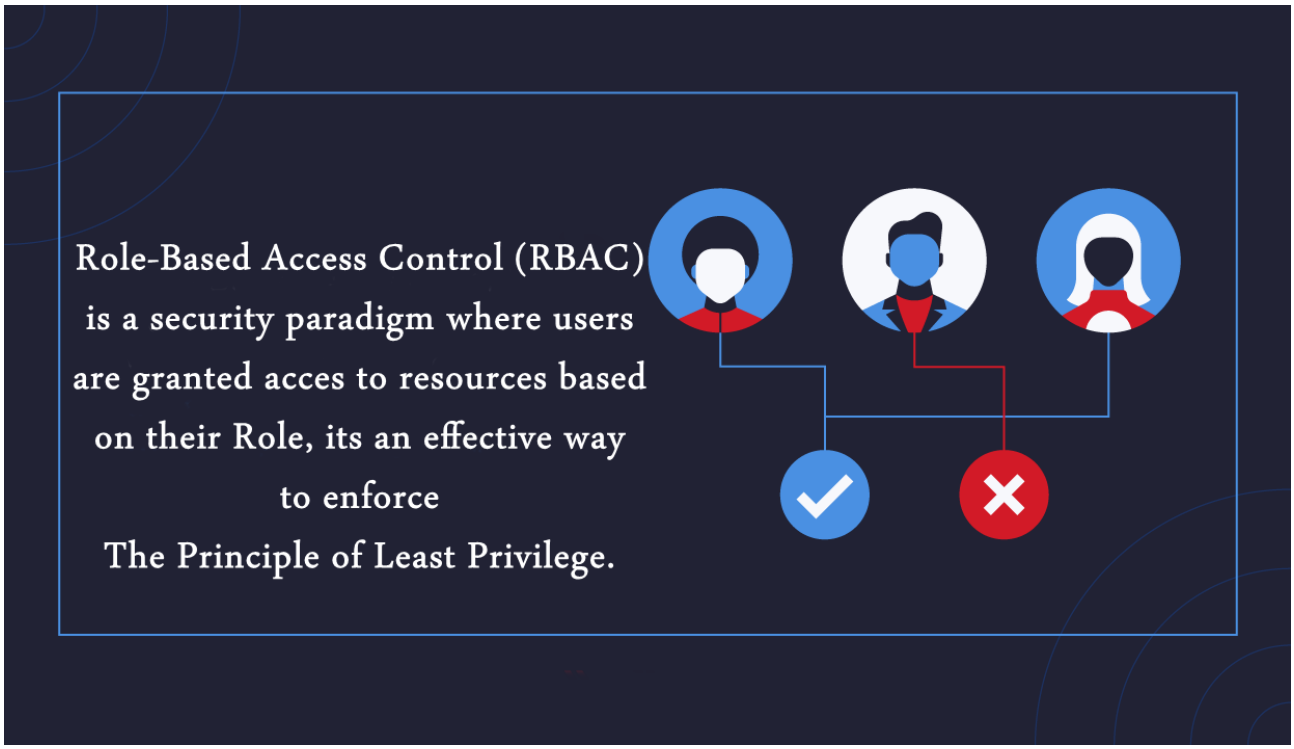


*Figure 103. Role-Based Access Control.*

According to Saltzer and Schroeder's design principles of Information Protections: Least privilege means that Every program and every user of the system should operate using the least set of privileges necessary to complete the job, thus in the implemented InMemoryDB system, the system only grants access to users within their scope of necessary rights of action based on their role. i.e., A viewer user cannot add or remove records because its beyond his scope of actions.

With every user having his unique email and password and a role we can enforce the RBAC, the user will be directed to his home page after a successful login, so a viewer user cannot access an editor homepage.

- **Password Encoding**

Passwords are not stored as a plain text in the user's database, passwords are stored as SaltedPassword+Hash in the traditional Servlets/Jsp version and as SYcrypt in the Spring version. That way it will be a hard task for an intruder to brute force the passwords.

a user cannot sign in and access the database system unless the given password matches with the one stored in the database.

In the traditional Servlets/Jsp version of the project, a SHA-512 Cryptographic Hash Algorithm is used here to hash the password and salt, SHA-512 is a very secure, and trust worthy algorithm.

```
public byte[] getSalt()   {
    SecureRandom random = new SecureRandom();
    byte[] salt = new byte[16];
    random.nextBytes(salt);
    return salt;
}
```

*Figure 104. getSalt Method.*

This method will generate a random salt value, SecureRandom class is used to generate good salts, the salt size is 16 bytes if its less than that an attacker may precompute a table of every likely salt appended to every likely password. Using a long salt will ensure such a table would be excessively large. Also, the salt is random because using the same salt for all passwords is dangerous.

Each user will have his own unique salt, so if 2 users have the same password their hashed Password will be completely different.

```
@Bean
DaoAuthenticationProvider authenticationProvider(){
    DaoAuthenticationProvider daoAuthenticationProvider= new
DaoAuthenticationProvider();
    daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());
    daoAuthenticationProvider.setUserDetailsService(this.userServiceImpl);

    return daoAuthenticationProvider;
}

@Bean
PasswordEncoder passwordEncoder(){
    return new SCryptPasswordEncoder();
}
```

*Figure 105. Spring Security Config Methods.*

In the spring version of the project, I provided the beans of password encoder and authentication provider to the security config class. We configured the authentication provider to automatically use SCrypt by declaring the password encoder bean, and the user service implementation class which is connected to the User DAO.

SCrypt which is a password-based KDF, this algorithm was specifically designed to make it costly for intruders to perform large-scale custom hardware attacks by requiring large amounts of memory.

14 STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS

| KDF | 6 letters | 8 letters | 8 chars | 10 chars | 40-char text | 80-char text |
|---|---|---|---|---|---|---|
| DES CRYPT | < $1 | < $1 | < $1 | < $1 | < $1 | < $1 |
| MD5 | < $1 | < $1 | < $1 | $1.1k | $1 | $1.5T |
| MD5 CRYPT | < $1 | < $1 | $130 | $1.1M | $1.4k | $1.5 × $10^{15}$ |
| PBKDF2 (100 ms) | < $1 | < $1 | $18k | $160M | $200k | $2.2 × $10^{17}$ |
| bcrypt (95 ms) | < $1 | $4 | $130k | $1.2B | $1.5M | $48B |
| scrypt (64 ms) | < $1 | $150 | $4.8M | $43B | $52M | $6 × $10^{19}$ |
| PBKDF2 (5.0 s) | < $1 | $29 | $920k | $8.3B | $10M | $11 × $10^{18}$ |
| bcrypt (3.0 s) | < $1 | $130 | $4.3M | $39B | $47M | $1.5T |
| scrypt (3.8 s) | $900 | $610k | $19B | $175T | $210B | $2.3 × $10^{23}$ |

*Table 4. Estimated hardware cost to crack hashed password in an average time of 1 year.*

While I was researching and comparing hashing algorithms, I came across this table from "stronger key derivation via sequential Memory-hard functions" Scientific paper written by Colin Percival on ResearchGate.

In Table 4, he showed the estimated costs of cracking hashed password in USD, the cost of hardware which can find a password in an average time of one year. These values are estimated and reflect only the cost of the cryptographic circuity. Its quite possible that the costs of other hardware (control circuitry, boards, power supplies) and operating costs (power, cooling) would increase the costs by a factor of 10 above the estimated costs on the table.

But since its an old estimate study the costs might be much lower than that. Nevertheless. I used this table because the estimates presented in the table are useful for the purpose of comparing different KDF and why I chose SCrypt.

As we can see, its clear from that table that SCrypt is much more expensive KFD to attack than any other alternative in the table because SCrypt requires higher recourse demand and increases the memory and parallelism required to crack the password, which can significantly increase the number of resources required to crack it.

- **Web Filter**

A Java Servlet filter is used to intercept all the client request and to do some pre-processing to check if the client is authorized to access the requested page. We can use servlet filters to our advantage by intercepting and altering the request before they are sent to the client.

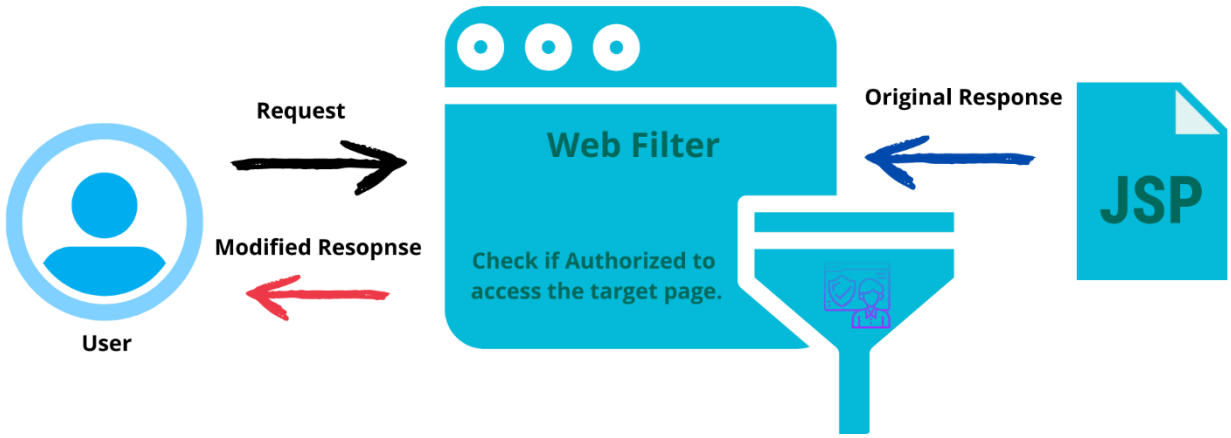


*Figure 106. Web filter diagram.*

Here, we are using the Web filter as an authentication and authorization of request for resources tool.

i.e., if a user with a Viewer role tried to access Admin/Users/AddUser page his request will be modified and he will be redirected to his homepage. Also, if a user is not logged in and tried to access Viewer/Books page he will be redirected to the login page.

```
@Override
public void doFilter(ServletRequest serverReq, ServletResponse serverRes,
FilterChain filterChain) throws IOException, ServletException {

    request = (HttpServletRequest) serverReq;
    role = (Role) request.getSession().getAttribute("role");

    if (role == null){
        forwardToLoginPage(serverReq,serverRes);
    }
    else {
        handleUserRequest(serverReq, serverRes, filterChain);
    }
}
```

*Figure 107. do Filter method.*

The doFilter() method will be invoked every time when a user request to access any resource. This way we can ensure that no unauthorized users or intruder can access data beyond their scope of authority.

- **Spring Security**

In the spring version of the project, we are using form based authentication which is the process of authenticating a user by presenting a custom HTML page (login page) that will collect user credentials and by directing the authentication responsibility to the web application that collects the form data.

So, the application here is responsible for dealing with form data and performing the actual authorization phase. Form based authentication is considered to be the most widespread form of authentication.

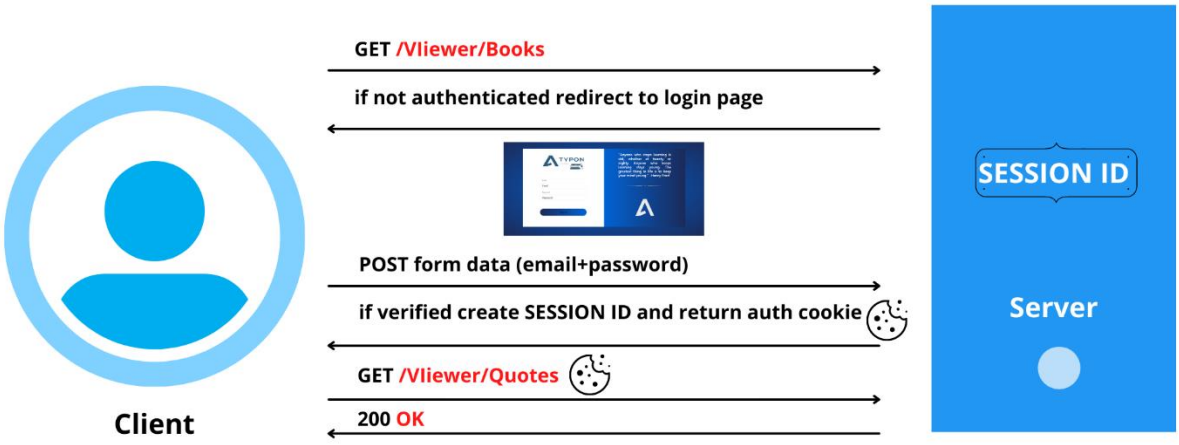


*Figure 108. Implemented Form based authentication.*

If the user credentials are correct a new session will be created by the web application and that session will have a unique session id. Also, the server will send back to the client a cookie containing the new session id that was just created. So, if the client sent another request to the server, the authentication cookie will be sent too to the server.

The server will check the session id against the sent session id from the client, if its valid and not expired then the request will be processed.

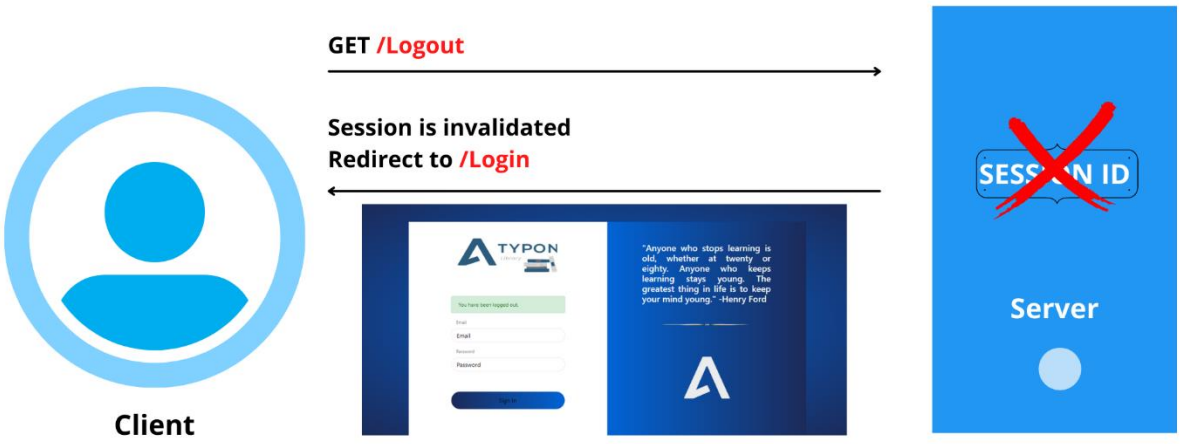


*Figure 109. Implemented Form based authentication 2.*

In case if the user wants to logout, the session will be invalidated and the user will be redirected to the login page. Also, the exact same process happens when the session expired after a period of time (default is 30M).

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http
            .authorizeRequests()
            .antMatchers("/login.html").permitAll()
            .antMatchers("/error.html").authenticated()
            .antMatchers("/Admin/**").hasRole("ADMIN")
            .antMatchers("/Editor/**").hasRole("EDITOR")
            .antMatchers("/Viewer/**").hasRole("VIEWER")
            .and()
            .formLogin()
            .loginPage("/login").usernameParameter("email")
            .successHandler(loginSuccessHandler)
            .failureUrl("/login?error=true")
            .permitAll()
            .and()
            .logout()
            .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
            .logoutSuccessUrl("/login?logout=true")
            .deleteCookies("JSESSIONID");
}
```

*Figure 110. Configure method.*

Behind the scenes, where configure method is doing all the magic we have implemented, we also need antMatchers for multiple Http Security, we use them here to apply authorization to many paths. antMatchers order is very important.

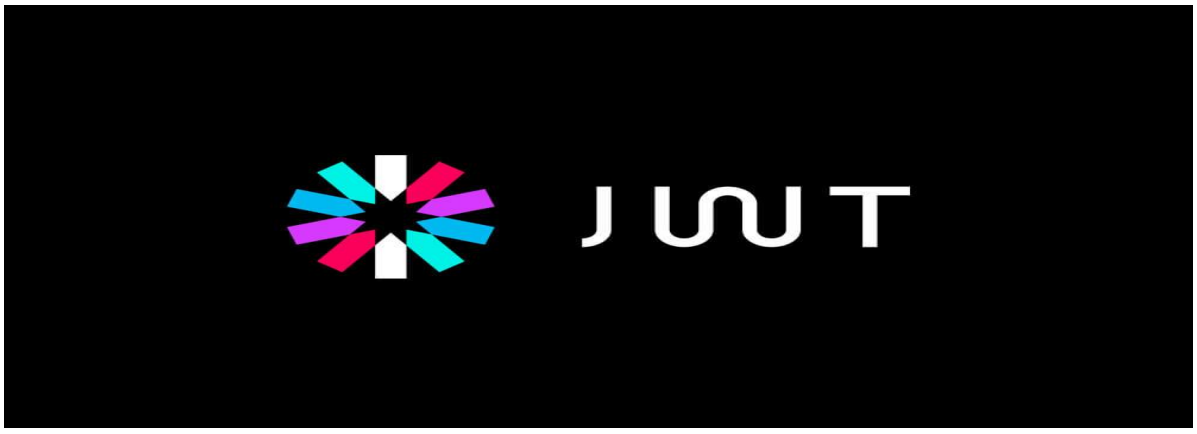


*Figure 111. JSON Web Token.*

Also, JWT is worth mentioning here, but since the implemented web application provides business logic and all the necessary user interface, and the system is simple and self-contained we don't need JWT because form based security will work fine.

But if the web application was more complex, having public **REST** endpoints given to third party apps (i.e., Facebook, google) or customers then JSON web tokens will be the best choice here.

JWT is a compact and safe way to transmit data between two parties, the information can be trusted because it is digitally signed and its consisted of three parts separated by dots:

1. Header.
2. Payload.
3. Signature.

- **SSL Certificate and HTTPS**

No web application is safe unless we have SSL certificate because the data and user credentials that is in transit can be easily intercepted and decoded, since we send the user credentials to the web application, we shouldn't send them in the clear, because if we do, we will compromise users' data to intruders.

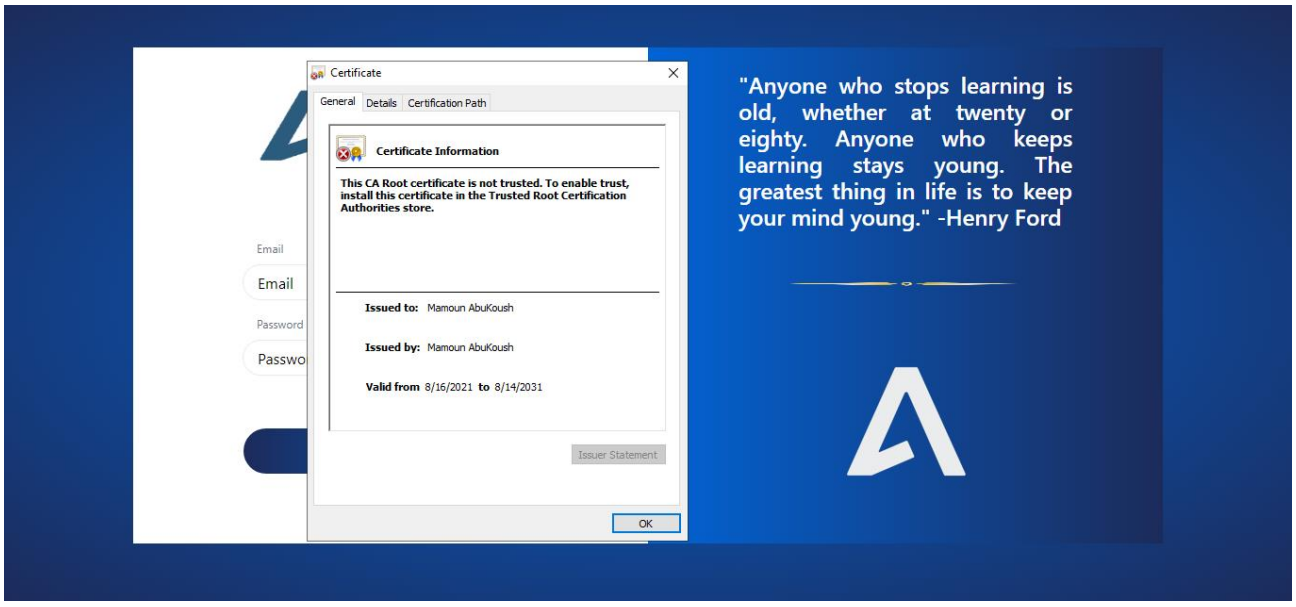


*Figure 112. SSL certificate.*

The SSL certificate is not trusted because it's issued by me not by a certificate authority, I did that only for development purposes. it will secure the communication by encrypting all data with session key.

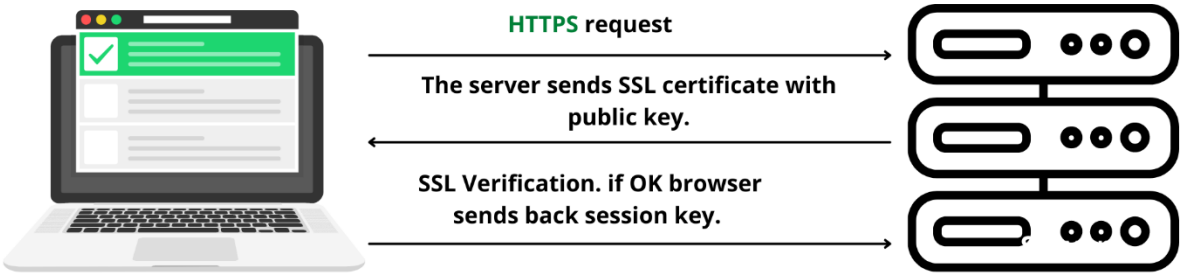


*Figure 113. SSL flow.*

- **Thymeleaf with Spring Security**

Views should be also security aware, in our web application project we are using Thymeleaf extras with spring security integration to show/hide content if user is authenticated or not. Also, to show/hide content if user has a specific role or permission.

```
<div sec:authorize="hasRole('ROLE_ADMIN')"  th:fragment="AdminNavBar">
<div sec:authorize="hasRole('ROLE_VIEWER')" th:fragment="ViewerNavBar">
<div sec:authorize="hasRole('ROLE_EDITOR')" th:fragment="EditorNavBar">


<a sec:authorize="hasRole('ROLE_ADMIN')" th:href="@{/Admin/Users}">Go To
Homepage</a>
<a sec:authorize="hasRole('ROLE_EDITOR')" th:href="@{/Editor/Books}">Go To
Homepage</a>
<a sec:authorize="hasRole('ROLE_VIEWER')" th:href="@{/Viewer/Books}">Go To
Homepage</a>
<a sec:authorize="isAnonymous()" th:href="@{/login}">Login</a>
```

*Figure 114. Thymeleafe with spring security.*

## 3.10 Front-End Design

The front-end design is simple and neat, for template engines in the traditional Servlet version of the project I have used JSP and in the Spring version I have used Thymeleaf. HTML, CSS and Bootstrap, JavaScript was used in the front-end design.

- **Traditional Servlet/Jsp version:**

```
<welcome-file-list>
    <welcome-file>login</welcome-file>
</welcome-file-list>
```

*Figure 115. Code Snippet from web.xml file*

As we can see in the above figure, I have defined the welcome file in the web.xml file. We can define so many things in the web.xml such as the mappings between URL paths and the controllers that handle request of those paths. Also context.xml file to define resources and other settings.

JSP is a java view technology which allow us to write template text in the client side (HTML, CSS, JavaScript). If we take a look into the JSP pages we would find that Scriptlets were totally avoided for many reasons. instead, we are using the well-known taglib JSTL which increases the readability and maintainability of the code.

```
<c:forEach items="${list}" var="book">
<td>${book.value.ID}</td>
<td>${book.value.name}</td>
<td>${book.value.author}</td>
<td>${book.value.subject}</td>
<td>${book.value.publisher}</td>
<td>${book.value.year}</td>
    <td><a class="btn btn-info"
href="/Editor/UpdateBook?ID=${book.value.ID}">Update Book</a></td>
<td><a class="btn btn-danger" href="/Editor/DeleteBook?ID=${book.value.ID}"
       onclick="if (!(confirm('are you sure you want to delete this book')))
return false"
>Delete Book</a></td>
```

*Figure 116. JSTL predefined tag.*

As we can see the using JSTL tags is very close to java and more readable than scriptlets.

- **Spring version:**

Using Thymeleaf made the job easier to me in this version of the project because it has more powerful syntax than JSP and way better integration with Spring. Also, one major benefit is high reusability, i.e., we created the user's navigation bar as a small template fragment and reused them in other templates.

```
<div th:fragment="header">

<div sec:authorize="hasRole('ROLE_ADMIN')" th:fragment="AdminNavBar">

<div sec:authorize="hasRole('ROLE_VIEWER')" th:fragment="ViewerNavBar">

<div sec:authorize="hasRole('ROLE_EDITOR')" th:fragment="EditorNavBar">

<div th:fragment="footer">
```

*Figure 117. Small template fragments.*

- **Error mapping and exception handling:**

In case of errors such as requesting a page that does not exist or an exception the web application will not send the user to a white label page (Default error page) instead the user will be redirected to an error page with the status code so the user can know what happened. Using a custom error page made the user interface design more friendly.

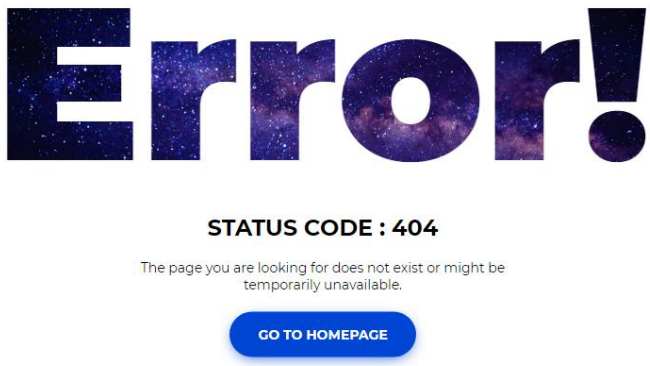


*Figure 118. Error Page.*

We implemented the error handler by using an error controller that will listen to exceptions/errors and direct the user to the error page with the status code.

## 3.11 Least Recently Used Cache

To minimize disk reads and store heavily used data in memory an efficient, light-weight and generic in-memory LRU Cache was implemented, it will cache 30% of the InMemoryDB.

```
public class LRUCache<K, V> implements Iterable<LRUCache.Node<K, V>>{

    public static class Node<K, V>{
        private K key;
        private V value;
        private Node<K, V> prev, next;

        private Node(K key, V value){
            this.key = key;
            this.value = value;
        }

        public K getKey() { return key; }

        public V getValue() { return value; }
    }


    private class LinkedList {
        private Node<K, V> head, last;
```

*Figure 119. LRU Cache Class*

The LRU cache consists of a doubly linked list and a hash map. Each node in the linked list holds a key-value pair in the cache. Whenever a key is retrieved, either by an update or a read operation, the corresponding node is sent to the head of the list.

By that we ensure that the nodes are ordered by their access time with the most recently used nodes at the head of the list and the least recently used at the tail. The cache keeps updating after every operation from the user.

# 4 Testing

## 4.1 Introduction

According to Michael feathers in his book working Effectively with Legacy Code he defines a legacy code as a code without tests. If someone wants to know what the code is doing then he will play the computer in his head and start thinking about every possible scenario and that's what I like to do, or test manually to see what it does and how it acts against different inputs and scenarios.

There's no excuse to not write tests, many test frameworks out there, in this system I used **JUnit** testing framework and manual testing.

```
@Before
public void setUp() { inMemoryDatabase = InMemoryDatabase.getInstance(); }

@Test
public void testConcurrentOps(){
    Map<String, Book> books;
    try {

        // thread to add record
        new Thread(new Runnable() {...}).start();

        // thread to add record
        new Thread(new Runnable() {...}).start();

        // thread to read all records
        new Thread(new Runnable() {...}).start();
    } catch (Exception e) {...}

    assertTrue(inMemoryDatabase.recordExists( id: "TestNum1"));
    assertTrue(inMemoryDatabase.recordExists( id: "TestNum2"));

    inMemoryDatabase.remove( id: "TestNum1");
```

*Figure 120. JUnit Test Example from InMemoryDBTest Class*

## 4.2 JUnit and Manual Tests

JUnit test was implemented to test the correctness of the implemented structure of several classes, also clean code principles apply to tests. The tests are simple and expressive, tests were run using **Maven**. In the bellow tables ill describe the unit and manual testing that I did to test the system.

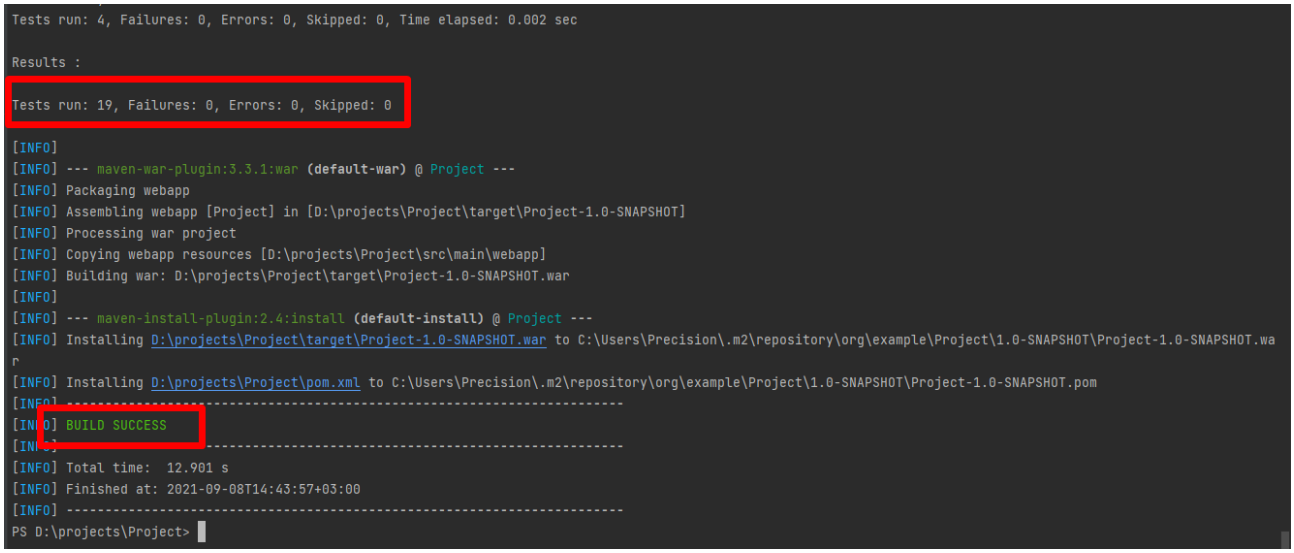


*Figure 121. Maven Build.*

- **Login Test:**

| Input Values | Expected Results | Pass/Fail | Results |
|---|---|---|---|
| Valid Credentials | Login Successfully. | Pass | System allowed Valid user to log in and showed the homepage based on his role. |
| Invalid Credentials | Display error (E.g., “Invalid Credentials”) | Pass | System displays invalid user message user and asks him to try again. |

*Table 5. Login Test.*

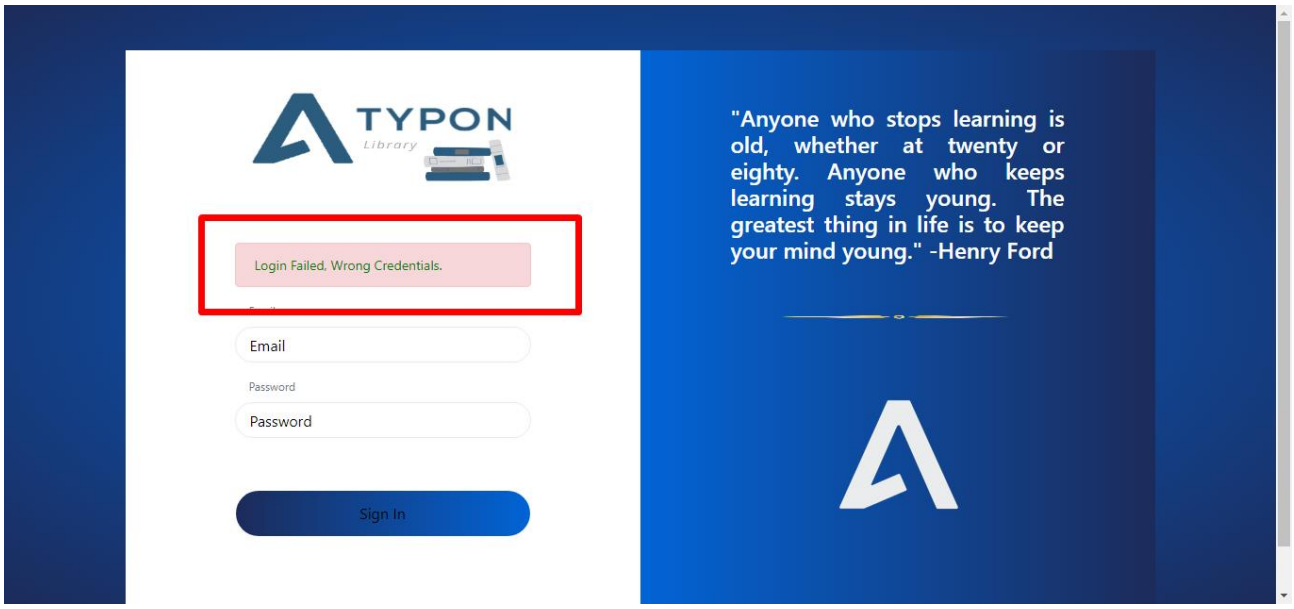


*Figure 122. Failed login.*

- **In-Memory DB Test:**

| Input Values | Expected Results | Result | Pass/Fail |
|---|---|---|---|
| 3 Threads preforming Book update transactions at the same time. | Concurrent execution with no problems | Threads were able to complete their tasks successfully. | Pass |
| 3 Threads preforming Quote update transactions at the same time. | Concurrent execution with no problems | Threads were able to complete their tasks successfully. | Pass |
| Get an existing Book/Quote by id. | Requested record is returned. | Record returned successfully. | Pass |
| Get all records. | All records are returned. | Records returned Successfully. | Pass |
| Add new Book/Quote | Record is added. | Record is checked and added successfully. | Pass |
| Update Book/Quote | Record is updated. | Record is updated successfully. | Pass |
| Remove an existing record. | Remove record from In-MemoryDB and log. | System removes the record successfully | Pass |
| Update an existing record. | Update record and inform user. | System informed the user that the record has been updated | Pass |

*Table 6. In-MemoryDB Unit Tests.*

- **UserDB Test:**

| Input Values | Expected Results | Result | Pass/Fail |
|---|---|---|---|
| Create User with unique user name | User Account is created. | New account was created for the user and stored into users db. | Pass |
| Remove an existing user | Remove user from users db | System removes the user successfully | Pass |
| Add a user with an existing username | Display “user already exists” | System informed the Admin that a user holds the same username. | Pass |

*Table 7. UserDB Unit tests.*

- **Input Exceptions Test:**

| Input Values | Expected Results | Result | Pass/Fail |
|---|---|---|---|
| Null Object | Return True | Returned True | Pass |
| Valid Object | IsNull returns false | Returned False | Pass |
| Empty String | Return True | Returned True | Pass |

*Table 8. Input Exceptions Unit Test*

Many more test cases were carried out to test the whole structure of the system, below are some of these tests:

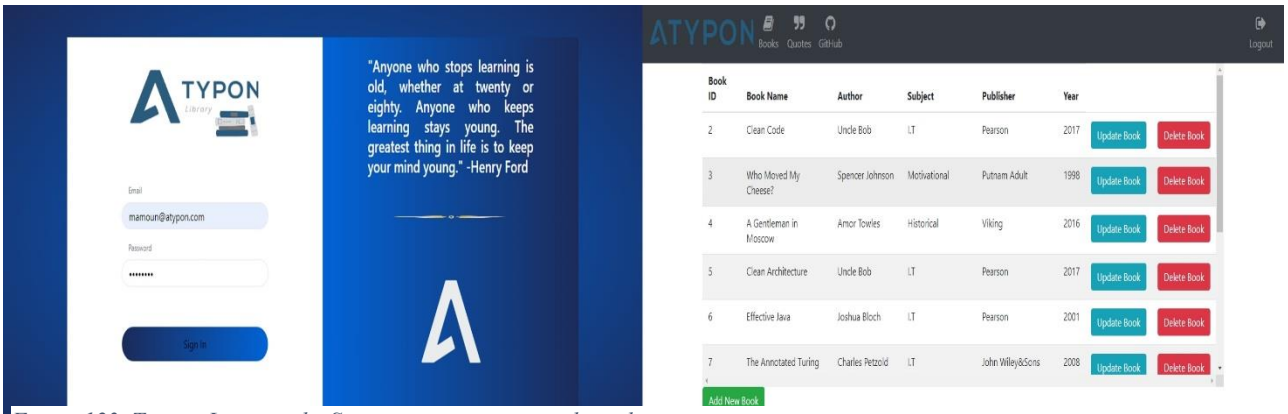


*Figure 123. Testing Login to the System using correct credentials*

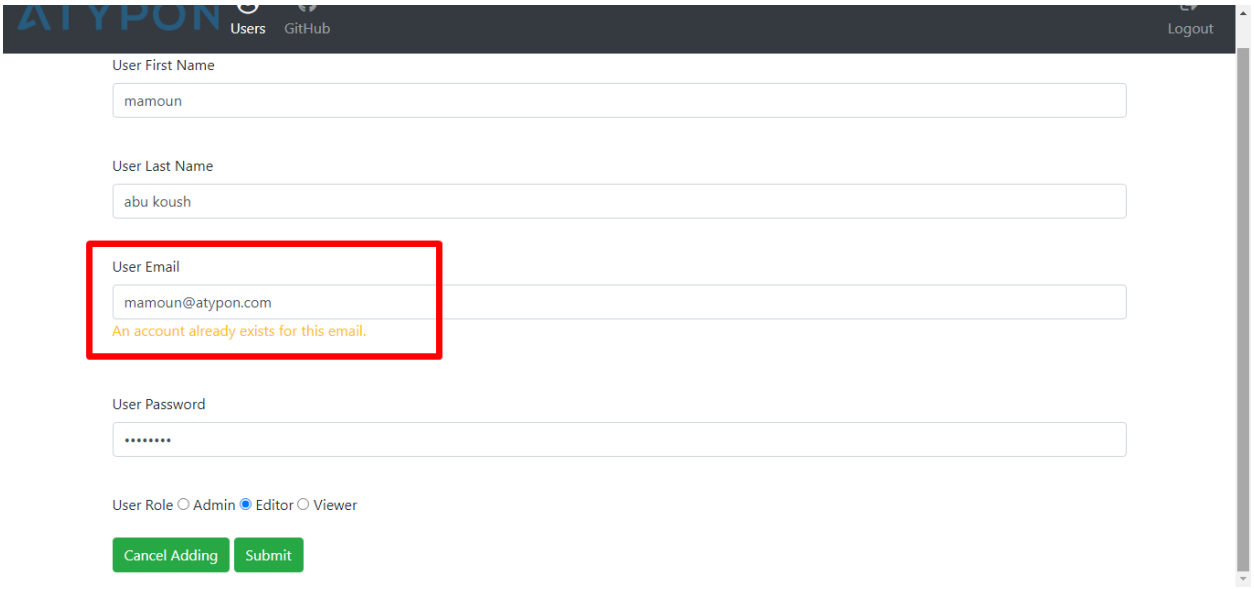


*Figure 124. Testing adding an existent user to the system.*

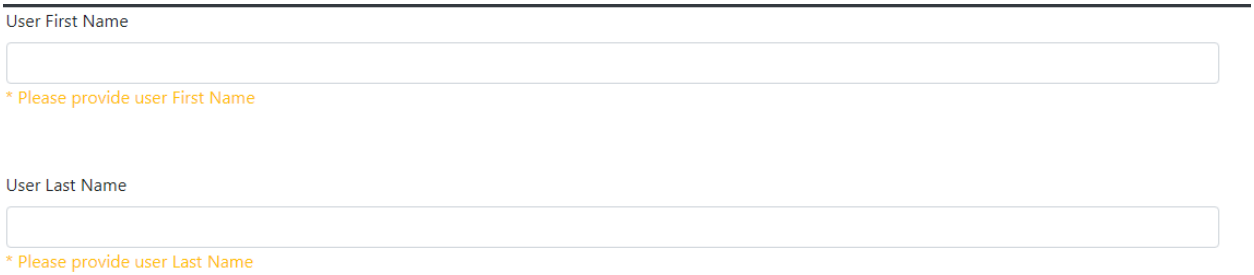


*Figure 125. Testing adding a new user with empty values.*


*Figure 127. Testing adding a new user with invalid email.*

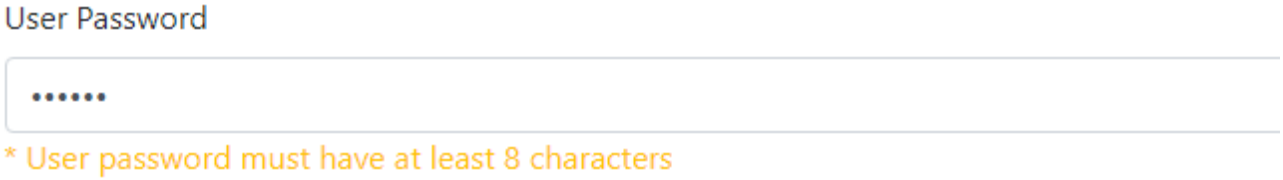


*Figure 126. Testing adding a user with weak password.*

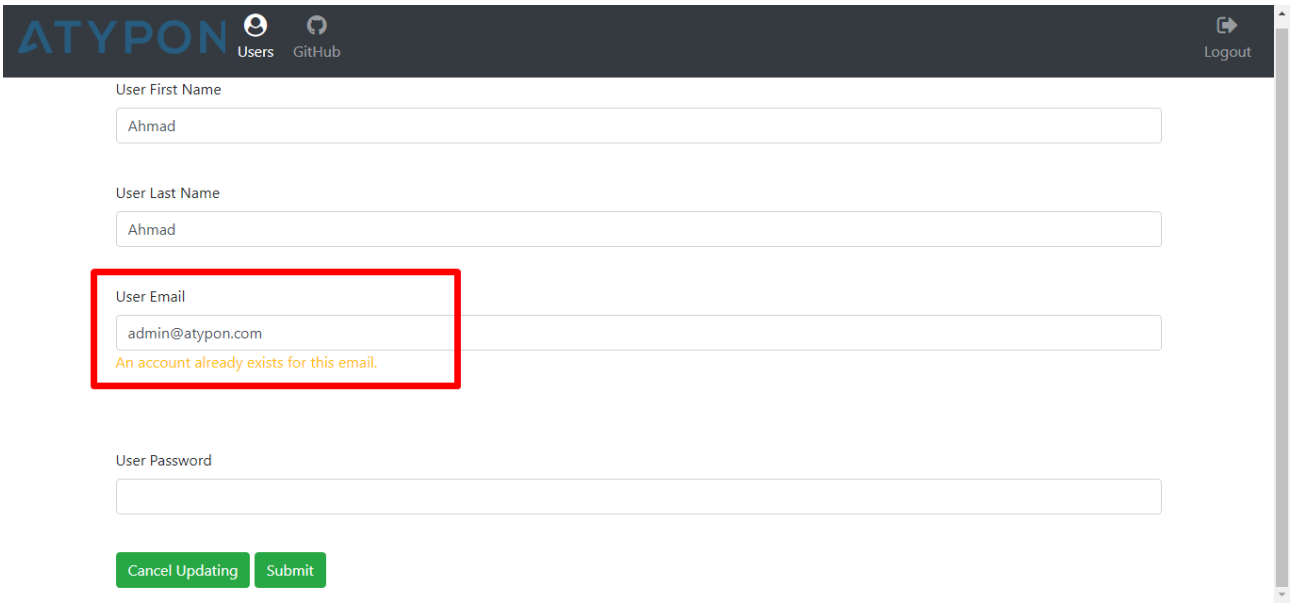


*Figure 128. Testing updating an existing user with another existing user email.*

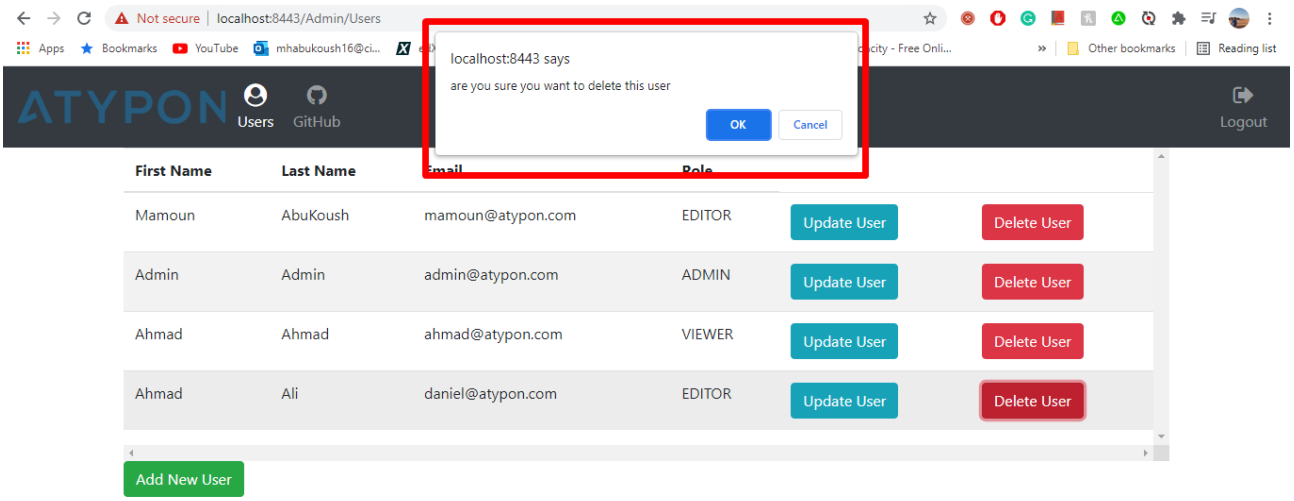


*Figure 129. Testing deleting a user.*

The same thing goes too in case of adding, deleting and updating Books or quotes, the same constraints and validation checks are applicable to them too.

## 4.3 Login Credentials to use the system

- To login to the system please use one of the following users and passwords:

| Username | Role | Password |
|---|---|---|
| **admin@atypon.com** | Admin | 12345678 |
| **mamoun@atypon.com** | Editor | 12345678 |
| **ahmad@atypon.com** | Viewer | 12345678 |

*Table 9. Login Table.*